MASTER IN COMPUTER SCIENCE

On secure services for embedded mobile peer-to-peer systems

a conceptual framework and its implementation in the coordination language SecureLime

Buchholz, Jean-Louis; Lange, Julien

Award date:
2008

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique

Année académique 2007-2008

On Secure Services for Embedded Mobile Peer-to-peer Systems: a Conceptual Framework and its Implementation in the Coordination Language SecureLime

Jean-Louis Buchholz,
Julien Lange.

Mémoire présenté en vue de l'obtention du grade de maître en informatique.

Résumé

Le développement récent des communications sans fil à courte portée a permis l'émergence d'une nouvelle topologie de réseaux, connue sous le nom de MANET, signifiant "Mobile Ad Hoc Networks". Un réseau ad hoc est formé par des appareils hétérogènes et potentiellement mobiles qui sont reliés entre eux par des connexions sans fil à courte portée. Ces appareils sont considérés comme des *peers* parce qu'aucun d'entre eux n'a de rôle pré-assigné. En effet, ils ne peuvent pas compter sur une infrastructure pré-existante telle qu'un serveur centralisé, par lequel ils pourraient interagir. Le développement d'applications sur les réseaux ad hoc soulève de nombreux problèmes. En particulier, l'inhérente hétérogénéité des appareils formant les réseaux ad hoc requiert le développement d'un protocole de communication générique que tout peer doit utiliser. De plus, le manque de fiabilité des peers demande de nouveaux moyens pour gèrer la dynamicité de leurs connexions. Enfin, les réseaux ad hoc offrent typiquement une résistance plus faible aux attaques de sécurité à cause de l'utilisation de communications sans fil.

Définir un middleware cachant la complexité induite par ces réseaux est l'approche classique pour faciliter la tâche des programmeurs d'applications ad hoc. Le découplage en espace et en temps fourni par le modèle des espaces de tuples (tuple spaces) apparaît comme un moyen élégant et efficace pour développer un tel middleware. Ce mémoire vise justement à expérimenter le développement d'un middleware ad hoc en utilisant un système de coordination basé sur le modèle des espaces de tuples. Pour atteindre ce but, nous procédons comme suit. Premièrement, nous décrivons le projet européen SMEPP qui a pour objectif de développer un middleware ad hoc équipé d'un langage de spécification, appelé SMoL. Deuxièmement, dans le but de choisir le meilleur système de coordination sur lequel développer le middleware SMEPP, nous étudions un large ensemble de modèles de coordination en tenant compte des exigences de SMEPP. Troisièmement, nous proposons un cadre conceptuel et son implémentation dans le langage de coordination SecureLime. L'implémentation est constituée de deux parties: une API pour le middleware et un traducteur de spécifications SMoL. Finalement, nous apportons un regard critique sur notre prototype. Ce nouveau regard ouvre, entre autres, de nouvelles perspectives de travail.

Abstract

The recent advances in short distance wireless communication resulted in the emergence of a new network topology, known as *MANET*, which stands for mobile ad hoc networks. An ad hoc network is formed by heterogeneous and potentially mobile devices bound together with short range wireless connections. These devices are considered to be *peers* because none of them has a statically assigned role. Indeed, they cannot rely on a pre-existing infrastructure such as a centralised server to interact with each other. Ad hoc networks raise many issues to be faced when building applications on top of them. In particular, the inherent heterogeneity of the involved devices requires a generic way of communication to be used by all peers. Furthermore, the lack of peers' reliability demands new means to handle highly dynamic connections between them. Finally, ad hoc networks are usually weaker against security attacks because of the use of wireless communications.

Defining a *middleware* hiding the complexity induced by these networks is the classical approach to ease the programmer's task. The space and time uncoupling provided by the *tuple space model* appears as an elegant and effective mean to develop such a middleware. This thesis actually experiments the development of an ad hoc middleware on top of a tuple space based *coordination system*. To achieve this goal, we proceed as follows. Firstly, we describe the SMEPP European project which intends to develop an ad hoc middleware which is equipped with a specification language, called SMoL. Secondly, in order to choose the most suitable coordination system on top of which we can build the SMEPP middleware, we survey a broad set of coordination models with regards to the SMEPP requirements. Thirdly, we propose a conceptual framework and its implementation in the coordination language *SecureLime*. The implementation consists in two components: a middleware API and a SMoL translator. Finally, we bring a critical point of view on our prototype implementation which notably opens new work perspectives.

Acknowledgements

We would like to thank the people who supported us throughout the writing of this thesis.

Especially, we thank Prof. Jean-Marie Jacquet (University of Namur) for his support in fixing the objectives of this thesis and for all his helpful advice and remarks.

We also want to thank Prof. Antonio Brogi (University of Pisa) for his full support and warm hospitality during our internship in Pisa and for all his advice during our first experience in scientific writing.

Furthermore, we are very grateful to Fabrizio Benigni and Razvan Popescu (University of Pisa) for all the interesting discussions we had together and their advice during our work.

Finally, we would like to thank our respective families for the support they gave us during our whole studies.

Contents

List of Figures

Chapter 1

Introduction

Peer-to-peer systems have become more and more common as a solution for a wide range of applications. The emergence of such systems brought a completely new application architecture where the usual hierarchy between application components is no longer relevant. Indeed, the client/server model was the classical answer to distributed computing. Today, this strongly hierarchical model is not suitable anymore in many cases, particularly in the field of ad hoc networks.

The recent advances in short distance wireless communication opened up new areas of applications where mobile devices can collaborate using wireless channels [99]. Currently, these advances are highly used in the field of personal mobile devices. Indeed, today more and more people possess PDA or new generation mobile phones which feature wireless communication. Owners of such devices have induced new communication needs resulting in a massive development of heterogeneous applications trying to answer the demand. Furthermore, the appearance of electronic sensors featuring wireless communication induces similar needs, for instance, in the domain of home automation and environmental monitoring (especially in the field of nuclear energy [99]).

A consequence of these advances is the emergence of a new network topology, known as MANET [26] which stands for mobile ad hoc network. This kind of networks interconnects several mobile devices (PDA, mobile phones, routers, etc.) and is not based on a pre-defined infrastructure. For instance, a newly connected node does not know how the network is organised before effectively communicating with other nodes. Moreover, in such a network, nodes are free to join or leave it whenever they want. Thus, the network's wireless topology may change rapidly and unpredictably.

The traditional client/server model does not fit MANETs in the sense that it is structured in a strong hierarchical way. The peer-to-peer model [68] is more suitable for this kind of networks. Informally, peer-to-peer systems are based on the concept of resource sharing by direct exchange between nodes. Those nodes are symmetrical and thus do not rely on dedicated nodes to communicate. All kinds of resources can be shared: computing power, storage capacity, access to physical resources, etc. [22]. This paradigm is the opposite of the client/server one, where some powerful computers (servers) are used to provide services to a large number of clients.

This kind of networks features many advantages [11, 22, 99]. We outline a non-exhaustive list of the main advantages of ad hoc networks.

- **Flexibility.** Being based on non-predefined infrastructures, ad-hoc networks can adapt easily to change. If some peers disconnect or new ones appear, the collaboration can continue.

- **Independence.** Ad-hoc networks do not rely on servers or any central administration. In this way they avoid to have a single point of failure.

- **Self-healing.** Nodes can re-configure themselves to keep routing information updated, for instance. Each node is also in charge of his own resources (power, data, etc).
- **Scalability.** Each node contributes to the network capacity. One does not need additional resources to add a new node.
- **Low cost.** Getting started costs are really low, there is no need to install base stations. The creation of temporary setup is then easier.

The main drawbacks are also due to those characteristics. Firstly, being so flexible and independent, ad-hoc networks are often composed of heterogeneous softwares and hardwares. The challenge is to find a generic way of communication which could be used on all kinds of platforms. Another drawback is the lack of node reliability. Indeed, each node is free to leave the network whenever it wants or whenever it moves out of range. Moreover, such a network is usually weaker against security attacks because of the use of wireless communication. Often, this is worsen by the lack of computational power of the nodes to encrypt data. This requires, on the one hand, to define a way to manage efficiently topology changes, and, on the other hand, to secure the communications by taking into account the weak computational power. Finally, each peer must be programmed in such a way that it is able to autonomously manage, at run-time, its interactions and interconnections with its environment [22]. From the point of view of a node developer, this is an overhead compared to the use of a more classical client/server approach. To help the programmer in this task, one can think about defining a high-level language for programming these aspects.

Some approaches tried to provide a way to reduce the impact of the above mentioned drawbacks. JXTA [61] is the most famous one. JXTA offers protocols allowing a peer to publish services, to discover other peers and to open/close connections with other peers. Unfortunately, JXTA does not solve all the aforementioned drawbacks. It does help much to manage the dynamic topologies of the network. A better solution for this problem is the specification of a generic service model encompassing features which wrap the difficulties induced by ad hoc networks. This model has to be implemented in a specific middleware designed to work on heterogeneous hardwares and softwares.

As suggested in [51], "the specification of the internal behaviour of the components in a distributed computation or application should be distinct and separated from the specification of their interaction and dependencies. A coordination model defines the medium that the components exploit in order to coordinate, as well as the rules governing the interaction between the components and the coordination medium. On the other hand, a coordination language is a linguistic embodiment of a coordination model, i.e. it is the language that can be used to program the interaction among components according to the coordination model". The first and most representative example of this theory is Linda [50] which provides the basic high-level blocks of coordination. One can use those blocks, by extending and composing them, to build the aforementioned middleware. Despite the apparent elegance of this approach, it raises a fundamental challenge. Linda-like languages are based on the idea of a centralised data space which inherently comes into conflict with the definition of ad hoc networks. Indeed, a centralised unit does not make sense in this kind of networks. This challenge is the starting point of the research presented in this document, namely, to study the research made in the field over more than twenty years in order to find an extension of Linda which fits the requirements of ad hoc networks. The analysis of the state-of-the-art of coordination languages led us to select an extension of Lime, called SecureLime. The main features of this language are mobility (using transiently shared data space) and security (using symmetric keys) mechanisms.

In order to experiment the result of this research, we had the opportunity to take part of the SMEPP European project. SMEPP stands for Secure Middleware for Embedded Peer-to-Peer. "Its goal is to develop a middleware that will have to be secure, generic and highly customisable allowing for its adaptation to different devices (PDA, smart phones, embedded sensor actuator systems) and different domains (critical systems, comsumer entertainment or communication).

The key features of the middleware are groups of peers, service offered by peers or groups and security concerns. Furthermore, SMEPP is equipped with a high-level language (called SMoL) which allows to specify how to orchestrate a peer code. This language notably simplifies the time-consuming and error-prone task of specifying the interactions of a complex peer-to-peer system" [14].

Within the context of SMEPP, our contribution is to develop a proof-of-concept implementation of the middleware on top of a coordination language. This implementation has two goals. On the one hand, it evidences the feasibility of the tuple space approach in the ad-hoc network field. On the other hand, it provides a first prototype to simulate and experiment the SMEPP service model. The implementation contains an API for the SMEPP middleware and a translator from SMoL to this API. This approach then provides a reusable code generator from SMoL.

To reach our goals, the rest of our master thesis is structured as follows. Chapter 2 details the context in which the proof-of-concept takes place. It also provides more requirements on the coordination language to search for. Chapter 3 browses the state-of-the-art in the field of coordination models, keeping in mind the requirements elicited in Chapter 2. Chapter 4 describes the design and implementation of the proof-of-concept. It is subdivided into two main parts: the first one dedicated to SMoL translation concerns while the second one dwells on the middleware API. The latter has been published as a paper[1] to the *Workshop on Coordination Models and Applications: Knowledge in Pervasive Environments* in June 2008. Finally, Chapter 5 looks critically both at the implementation itself and at its context and foundations. This chapter also offers perspectives for future work.

[1] Available in Appendix C.

Chapter 2

Context

This chapter presents the context of our master thesis. As already mentioned in the previous chapter, we had the opportunity to work within the SMEPP European project which intends to develop a secure middleware for embedded peer-to-peer systems.

This chapter is structured as follows. Section 2.1 introduces the project and presents its key concepts. Then, Section 2.2 shows two typical applications which are expected to be implemented using SMEPP. This allows the reader to see concretely the target of SMEPP. Having a good overview of the project, Section 2.3 then presents the SMEPP service model which defines the basic communication blocks of the middleware. Section 2.4 gives the goals of our work in the SMEPP project. Finally, Section 2.5 drives the reader to the coordination language on top of which we can develop a proof-of-concept of the service model. In particular, it details the requirements that the language has to meet to make the implementation possible.

2.1 Overview of SMEPP

The purpose of the Secure Middleware for Embedded Peer-to-peer Systems (SMEPP) project is to develop a middleware which meets the new challenges raised by embedded peer-to-peer systems (EP2P) for distributed systems. These systems are extremely vulnerable against internal or external attacks (because of resource constraints and the nature of open communication channels). Moreover, application development of EP2P systems requires to compensate for the disappearance of centralised entities and pre-defined infrastructure. One needs to abstract all these problems by means of a convenient middleware. The main objective of SMEPP is to develop a new secure and generic middleware which could be used in many domains from home automation to nuclear plant monitoring.

SMEPP is a project funded by the European Union and is part of the "Sixth Framework programme" (FP6). FP6 intends to contribute to the creation of an "European Research Area" through the promotion of better cooperation between European actors. "Information society technologies" is the second main priority of the Sixth Framework. The actors involved in the SMEPP project are:

- Università di Pisa (Italy),
- Universidad de Málaga (Spain),
- Tecnatom, SA (Spanish engineering company),
- Technische Universitt Graz (Graz university of technology, Austria),
- Siemens AG (German company notably active in power, transportation, information & communication, medical and lighting),
- Valtion Teknillinen Tutkimuskeskus (Finnish technical research centre),

- Telefónica I+D (Spanish telecommunication company),
- Institute for Infocomm Research (information & communication research and development centre in Singapore).

In the rest of this section, we give an overview of the SMEPP project necessary to understand our work. Firstly, we overview the SMEPP service model which provides a framework for EP2P development. Secondly, we introduce the security aspect of the SMEPP project. Finally, we outline a specification language, called SMoL, which provides an abstract model of peer and service behaviour.

2.1.1 Service model overview

In order to build a generic, high-level and customisable middleware, an objective of SMEPP is to specify a high-level service-oriented model. This model provides a framework for the development of interactions between peers. The service model features a set of abstract primitives which can be used to develop peer-to-peer applications in a high-level manner. The basis of the model rely on the state-of-the-art of web service technologies. For instance, a service signature is modelled using the Web Service Description Language (WSDL [109]). Furthermore, services behaviours are specified similarly to what is achieved by the Business Process Execution Language (BPEL [78, 79]). The rest of this subsection describes the key concepts of the model.

Peers. A peer is the basic entity of a SMEPP application. Intuitively, it is a program interacting with other programs of its kind, i.e. other SMEPP peers. The communication is based on services which are offered by peers. One may say that peers are service containers.

Groups. A group is a logical collection of peers. It provides a secure communication environment and a scope for service discovery. Groups are created by peers and they are kept alive as long as they contain at least one member.

Services. A service is a set of functionalities offered by one (or several peers). Those functionalities, or operations, can be invoked by other peers to retrieve information or to execute a task. Note that a service is identified by its contract. This means that several peers can offer a same service while each instance is located in a different place. In the SMEPP service model, a service consists of two parts, its *implementation* and its *contract*. The contract provides descriptive information on the service while the implementation represents the executable service. Both parts of a SMEPP service are specified using the eXtensible Markup Language (XML). The contract is modelled by using WSDL and the implementation could be modelled by using the SMEPP Modelling Language (SMoL) or by any other language[1].

The contract itself is composed of two mandatory parts, the signature and the grounding. The signature describes "what the service does" by providing an abstract description of the operations included in the service. The grounding describes "how clients can invoke the service", i.e. it provides information on the procedure a client has to use to communicate with the service (e.g.: port number, protocols, etc.). Other information may be added to a contract such as QoS information, behaviour description[2], etc.

Once a service is published, it can be seen both as a service provided by a specific peer (its provider) or as a service provided by the group in which it has been published. When a service is called through its group, the middleware is in charge of transmitting the invocation to a specific peer offering the service. This abstraction allows to hide the fact that there are several providers of a same service.

Services are divided in two classes: *state-less* and *state-full* services. The former gathers services which do not keep track of the past interactions with callers (e.g. operations can be invoked in any order). The latter is the opposite: state-full services keep track of their

[1] In this case, it requires language-dependant wrappers to provide interoperability for the SMEPP middleware.
[2] Written in SMoL.

interactions with clients. Moreover, state-full services can be instantiated by several sessions. Further details concerning services are available in Section 2.3.

Communication abstraction. The SMEPP service model features three ways of communication. Peers and services can send and receive messages, events and faults.

- **Messages** are used to invoke operations. The service model offers two kinds of operation: one-way and request-response. The former requires only an input message (which could be empty). The latter requires an input message (possibly empty) and produces an output message (possibly empty) as a result of the operation execution. Both kinds of operation block the invoker until the provider has executed a specific action. Note that the possibility of using empty inputs allows the creation of operations without parameters, while using empty outputs permits synchronous operations which do not need to return any data.

- **Events** introduce non-blocking communication. Peers and services raise an event and continue their execution as soon as it has been signalled to the middleware. To receive an event, peers and services "listen" to events of their interest. Note that an entity has to subscribe to an event type before being able to receive it.

- **Faults** are the mechanism to communicate failures. Two fault types exist. On the one hand, the middleware raises fault when the user does not comply with the service model. For example, when an operation invocation fails because the caller and the provider do not belong to the same group, a fault is returned to the caller. On the other hand, an entity can explicitly return a fault as a result of an operation (in the case of a request-response operation).

2.1.2 Security aspects

As stated before, SMEPP defines a security framework to enable secure communication in an EP2P network[3]. SMEPP provides two domains of security.

- **Routing security** ensures that only authenticated SMEPP peers participate in the network of a specific SMEPP application. One can see peers of a SMEPP application as members of a default group which use a secret key to secure their communication.

- **Group security** concerns the access restriction to a group and the privacy of the communication inside a group. According to their credentials, peers are allowed or not to become a member of a group. Once member of a group, a peer uses a secret key (shared by other group members) which is used to secure the communication between groups members.

Furthermore, three security levels are defined. The first level is *level 0*, the "no security" one: access to the SMEPP application and groups are granted to all peers. The second level, *level 1*, is based on the assumption that all peers share a set of pre-shared keys. There are two types of keys: one is the key granting access to the SMEPP network, the other gathers the keys granting access to groups[4]. *Level 2* provides the greatest level of security. It is based on asymmetric keys: each peer has a set of private/public keys and a set of attributes (in the shape of certificates) granting access to groups.

The default security level of SMEPP is *level 1*. The credentials are included statically in the SMEPP application. This implies that the set of possible groups is known in advance. Note that the choice of the security level as default is notably justified by the basic assumption that most of the authenticated peers will well behave[5]. This means that attackers in possession of credentials of a SMEPP application are considered as exceptional cases.

[3] An EP2P network established by devices belonging to the same SMEPP application.
[4] Note that each group has an associated key.
[5] Note the detection of a bad behaviour of a peer is not (yet) defined in the SMEPP project.

2.1.3 SMoL

The SMEPP service model features the SMEPP Modelling Language (SMoL). The objective of this language is to provide a high-level specification tool to orchestrate the SMEPP primitives. It allows to build complex P2P programs while being able to reason formally on the behaviour of peers and services. For instance, [12] defines a transformational semantics of SMoL in terms of YAWL workflows [107, 114]. Roughly, SMoL features usual constructions which one can find in every programming language such as loops, conditional branches, etc. It also provides some coordination-oriented constructions (e.g. `InformationHandler`) which allow to make the most of the SMEPP primitives. SMoL is inspired by the Business Process Execution Language (BPEL [78]). The SMoL language is detailed further, in Section 2.3.2.

2.2 Application examples

Before going further in the description of the project, it is important to see in a concrete manner what are the objectives of the SMEPP project. This section describes two real-life applications in which the SMEPP middleware is intended to be used (these examples are described thoroughly in [97]). It is the occasion to make the link between the SMEPP key concepts and the future applications. It will also permit to elicit some requirements of the project.

Those two examples are typical applications of ad hoc networks, but such networks are used in many other fields. We outline some of them. Environmental monitoring is a classical application field for ad hoc networks. For instance, [19] describes a sensor network system to monitor vineyards. Ad hoc networks are also used as an emergency response in case of disaster. For instance, CodeBlue [63] is an architecture providing naming and discovery, robust-routing, and security services, specifically designed to address challenges met in case of catastrophe. Another emerging applications for ad hoc networks are e-health systems. E-health is an emerging field which intends to gather information and medicine technologies to better meet needs of patients and health-care professionals. MANETs are also spreading in the home environment through home automation. Many commercial systems already make a house “feel” what is happening inside the walls.

In the following examples, the first one is related to the e-health field and home automation, while the second one is oriented towards environmental monitoring and disaster recovery.

2.2.1 Home systems and mobile telephony

This example is based on an existing application, SequiTel, which is an e-health platform developed by Telefonica I+D [87]. This application provides telecare through a client/server architecture. A residential gateway has to be connected at home via ADSL to the Service Provider. Users can subscribe to different services depending on their needs (elders, pregnant women, diabetics, etc.). The current system provides videoconference (for tele-consulting), alarms handling, agenda management, home automation control and vital parameters reading (such as temperature, electrocardiogram, blood pressure, etc.).

The main drawback of this system is that if there is a connection problem between the residential gateway and the service provider, the user can not run most of the services. The main contribution of SMEPP is to allow family members and friends to be introduced as “care takers”, by using a peer-to-peer architecture linking together the “patient” and his/her family and friends. Moreover, the user trust and service perception will be significantly enriched and improved because the user relatives and friends may be directly caring for their elders, for instance, in many non-critical situations.

With such an architecture, both telecared users and their relatives can act as peers and can share services between them. They can create groups to share experiences or any helping materials. One can imagine those groups can gather people undergoing the same problems or situations (pregnant women, renal patient, etc). Members of a same family can create their own group with all their relatives and friends.

Two main requirements are elicited:

- the system has to be able to spread alarms through the peer-to-peer network (through different topologies),
- the middleware has to provide confidentiality, integrity and authenticity in transmitted data.

A set of services are proposed to be developed using SMEPP for SequiTel. The following examples will help to illustrate the specification of the SMEPP service model.

- A **Browser** to navigate in the network in which the user is. It allows to access groups, peers and their services. This browser provides two navigation modes. On the one hand, the user can navigate through the physical network. The user interface shows the peers with direct connectivity and the services they are offering. On the other hand, the browser provides navigation through groups showing all peers belonging to them and services offered to the group.
- **Alarm notification** to family/friends group provides a way to spread an alarm message to the peers in the family/friends groups. If there are no connected peers in the group, the system can send the alarm to the Telecare Center. Note this service needs to avoid an intruder to receive the alarm without belonging to the right group.
- **Emergency alarm from a mobile device**. Devices such as PDA or new generation mobile phone can include an emergency alarm generation function. In this way, users can send a S.O.S message wherever they are and this message can possibly include coordinates to know where the user is.
- **File Sharing** allows users to share videos with exercises, documentaries etc. through the peer-to-peer network. In the current system, users can only download videos from the central server. With the new system, a peer is able to download a file from several sources (as in systems like BitTorrent) using the bandwith more efficiently.
- **Tele-consulting**. SMEPP improves the videoconference system provided by the current system of SequiTel which does not have the quality required by telecared users and doctors. Moreover, the videoconference can be extended to every SequiTel user (not only doctors/supervisors). Users can also use different kinds of devices such as mobile phones and laptops.
- The current **Agenda** is improved by adding the possibility of reminders or appointments acknowledgement. If there is no online peer, a warning can be sent to the family/friends group.
- **Workflow**. A data file containing the vital parameters readings can be, for instance, sent to different peers, to the telecared user's nurse every day and to his doctor every month. Each user can write in the document to add comments or other modifications.
- A **Chat** application would allow any user to communicate with people in the same situation. Moreover, it could decrease the number of queries to specialists (doctors, nurses, etc). Other similar applications such as a forum could also be studied.

Figure 2.1 illustrates the scenario of an emergency alarm[6]. The telecared user John Smith is sitting in his living-room but does not feel well. He pushes his emergency button because he knows the system automatically warns the needed persons. Mr Smith is connected to Sequitel through his residential gateway (e.g an ADSL router). His son, Peter, is an assistant user, which means he is a "telecare provider". Both are connected to the Telecare Center. Once the warning arrives to Peter's phone, he can set up a video conference with his father to see what is wrong. When Mr. Smith pushes the button, the system determines the best person to warn. The alarm is spread through the peer-to-peer network as follows. If there is more than one peer connected, the warning arrives to the nearest peer (of all the peers connected to Smith's

[6]This scenario and the associated Figure are adapted from [97].

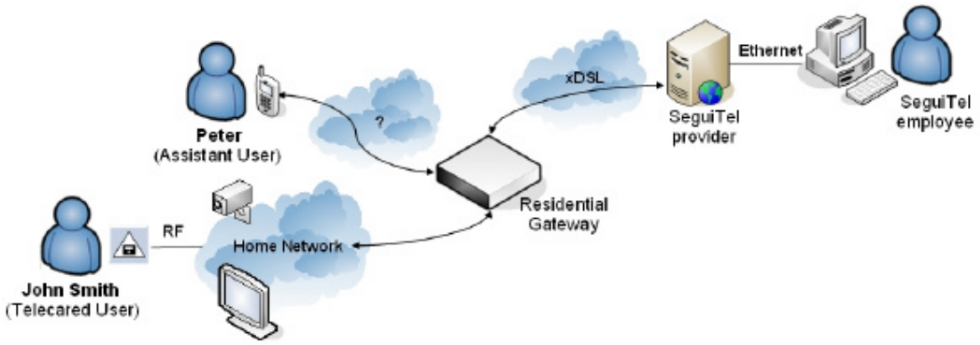


Figure 2.1: Sequitel example.

group). In case no peer is connected, the signal arrives to the Telecare Center, provided that the network is working.

Without a suitable middleware, the development of such a system would be tedious. Indeed, in addition to the application itself, three main issues would need to be faced. Firstly, the system will connect together different kinds of devices, such as mobile phones and desktop computers. Thus, without a middleware, the programmers would have to take care of the communication between devices from the physical layer. Secondly, since the application requires confidentiality, integrity and authenticity of the transmitted data, the programmers would also have to implement security protocols. Finally, programmers would have to face the transient connection of peers. Indeed, peers can join and leave the application whenever they want.

Through this first application example, one can easily link the key concepts of SMEPP with real life applications. For instance, each family member, friend and telecared user will be a SMEPP peer in the new version of SequiTel. One can imagine that a group can be created for each telecared user, gathering all his/her family members and friends. In addition, alarms and other notifications could be modelled using SMEPP events. Moreover, services such as file sharing or tele-consulting could be modelled using the concept of SMEPP service. Finally, in such an application, security has a significant role. Indeed, malicious peers using the system could endanger telecared user's life. Access to the system has then to be restricted.

2.2.2 Environmental monitoring and remote control of workers

This second application focuses on environmental monitoring and remote control of workers in industrial plants. Monitoring the effect of a plant on the environment is a key issue in different application domains, especially in the field of nuclear plants. Indeed, in the nuclear industry, the exposition to ionizing radiation is a risk present in daily operation and maintenance activities for workers present in the plant.

Current environmental monitoring and remote control of workers systems are based on a centralised architecture where a control room acts as a bottleneck of information and a potential risk if the communication is lost or if the control room itself is facing some troubles. The SMEPP middleware could provide a more robust and flexible architecture in which any "control node" could connect to the network in a secure way (from almost anywhere). Moreover, the peer-to-peer topology could provide a new way of communication inside team of workers.

Typically, the system will be composed of wireless sensor networks. Static radiation sensors and environmental monitors (measuring temperature, air quality, etc.) will be deployed outside and inside the plant to measure different environmental conditions. Furthermore, by regulation, current staff working in nuclear industry has to wear a so-called dosimeter which measures the degree of radiation received by a worker. In the proposed SMEPP application, every dosimeter will be connected to a small mobile device (such as a PDA or a mote) with two goals. On the one hand, it will enable a worker to monitor his/her dosimeter in a more advanced way. On the other hand it will provide a wireless connection to a control node, in order to send the current measurement. All those devices will form a large ad-hoc network which will allow a worker to

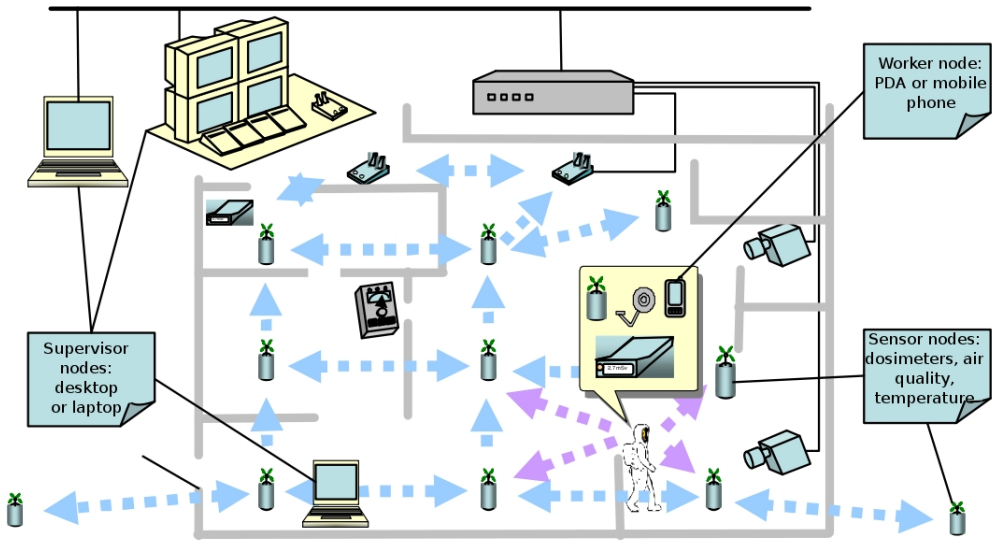


Figure 2.2: Plant monitoring example.

know the radiation level in the area where s/he is. This network will be monitored by a control node which will have a global view of the plant.

Three kinds of devices will form the system.

- Wireless sensors with compute capabilities to measure environmental conditions and/or radiation levels inside or outside the plant, for instance, temperature, air quality monitors, personal dosimeter, etc. Most of them will be tiny, low-cost and low-power devices but some of them can be bigger and more sophisticated. This kind of node will alert other devices as soon as they detect special conditions.

- Personal mobile devices (PDA, mobile phone etc.) connected to dosimeters worn by plant's staff. These devices allow a worker to check his/her personal radiation level as well as the radiation level present in nearby areas.

- Supervisor nodes are computers being able to monitor the whole network and supervise other workers. The software running on these computers will be able to display information about the network, to communicate with workers (through audio/video conference) and to dynamically set alarm levels in dosimeters.

Figure 2.2 (taken from [97]) shows a view inside the plant. Sensor nodes are deployed in the building and they are also carried by workers. Some workers have a PDA providing audio and video to interact with their co-workers. Furthermore, the figure shows several supervisor nodes that can interact with the system, both inside or outside the building.

The main requirements of the proposed system are:

- **Real-time.** Each device must be able to send information in (almost) real-time.

- **Devices.** The system can be composed of different kinds of devices and communication technologies. The middleware must provide heterogeneity in this way.

- **Confidentiality.** Provided the criticity of the domain, the data must be protected.

- **Absence of supervisor node.** The system must be able to work without a supervisor node.

- **Authentication.** Since nodes carried by workers will appear and disappear dynamically, the system has to provide a way to authenticate them automatically (i.e. without any intervention of the worker).

As in the first application example, the SMEPP team proposes several kinds of services to enrich the system. They are divided according to the kind of devices they are designed for. The following gives a non-exhaustive list of services which could be offered by using SMEPP.

- **Sensor nodes.**
  - *Data transmission.* Theses nodes must be able to transmit data to other nodes in the network.
  - *Transmission rate.* The nodes should allow supervisor nodes to configure transmission rate. For instance, in case of radiation sensors carried by worker, the rate may be one sample per second.
  - *Configuration.* Where possible, these sensors should be configurable by a supervisor node without physical contact to the device.
- **Worker nodes.**
  - *Alarm self-notification.* Worker nodes must notify their user about an alarm when the alarm level reaches risky conditions.
  - *Alarm notification to supervisor.* When a sensor detects abnormal measurement, it must be able to send an alarm to supervisor nodes.
  - *Alarm notification to nearby worker nodes.* When a sensor node (either static or carried by a worker) reaches an alarm level that can affect other workers, the system sends the warning to the active nodes of the nearby zones. It can then help to save seconds by warning a peer which is closer to the problem.
  - *Alarm reception from supervisor node.* The node will inform the user that an alarm is received from the supervisor that affects him/her.
  - *Alarm reception from nearby nodes.* This is the dual of the third services, the system must be clever enough to handle alarm in a efficient way (filtering low relevance ones for instance).
- **Supervisor nodes.**
  - *Browser* to navigate through the network and check other nodes status.
  - *Real-time dose/rate information* about workers and nodes.
  - *Programming alarm levels* in nodes.
  - *Automatic notification* to workers reaching alarm levels through audio or vibration.
  - *Worker management.* Self-identification, through dosimeter's identifier for instance.
  - *Environmental conditions map.* Map of environmental conditions inside and outside the plant.
  - *Time-out alarm.* When a node in the network does not transmit information for a certain period of time, an alarm is displayed to the user.
  - *Dose/Rate map* to show graphically the situation in the plant.

Similarly to the first example, the implementation of such an application is really hard without a suitable middleware. Notably because of the heterogeneity of the nodes (PDA, laptop/desktop and sensors) and the need for security. Furthermore, the system has to be very robust. It requires tried and tested means of communication between nodes.

Although this application is quite different from the first one, it still features common links with SMEPP key concepts. We find again the concepts of alarm and notification which can be modelled by events in the SMEPP service model. Remote configuration of nodes can be implemented by services, providing different operations for different types of configuration. Security is even more important in such a critical application.

2.3 SMEPP service model

2.3.1 Primitives

In order to provide a simple and high-level service model to ease the development of peer-to-peer application, SMEPP has defined a service model featuring a set of abstract primitives. Intuitively, those primitives are the basic blocks to define SMEPP programs (either a peer code or a service code). Figure 2.3 summarises them.

Note that some primitives can only be called by peer codes. Consequently, the middleware has to know whether the caller is a peer or a service (and, if needed, prevent the execution of the primitive). In that aim, each caller has an identifier and each type of entity has a different type of identifier. Peers identifiers are `peerId`s, services are identified by their `peerServiceId` and `groupServiceId` and sessions are identified by thier `sessionId`.

In the following, we shall use the following syntax to define the primitives.

`output name(input`$_1$`,..., input`$_n$`) throws exception exception`$_1$`, ..., exception`$_n$

Where	`name`	is the name of the primitive,
	`output`	is the type of the output parameter,
	`input`$_k$	is the kth input parameter,
	`exception`$_k$	is an exception throwable by the primitive,
	`par?`	denotes an optional parameter,
	`par[]`	denotes an array.

In the rest of this subsection, we present the SMEPP primitives as follows. We start by the peer management related primitives. Then, we present the primitives used to manage groups. The following part introduces the SMEPP service management and the corresponding primitives. The two last parts concern the communication by using messages and events.

Peer management

In the SMEPP service model, peer management is achieved by using three primitives. `NewPeer()` provides an operation for a program to become a SMEPP peer, `getPeerId()` allows a SMEPP entity to retrieve a peer identifier and `getPeers()` returns a list containing the identifiers of the members of a group.

NewPeer

`peerId newPeer(credentials) throws exception invalidCall`

To become a SMEPP peer, a program has to call the `newPeer()` primitive. This call will return a `peerId`, a peer identifier given by the middleware, provided the credentials authenticate actually the peer[7]. Otherwise, the `invalidCredentials` exception is raised[8] and the access to the SMEPP application is refused. Furthermore, only a peer code is allowed to call this primitive. When called by a service code, the primitive raises an `invalidCall` exception.

Example: *In the context of the plant monitoring application, this primitive may be used to authenticate nodes carried by workers. Thus, only nodes having the right credentials have access to the application.*

GetPeerId

`peerId getPeerId(id?) throws exception invalidId`

where `id` is either a `peerServiceId` or a `sessionId`.

If an `id` is specified, this primitive returns the identifier of the peer offering the service identified by `id`. Otherwise, the primitive returns the `peerId` of the caller peer. If the middleware cannot find an entity corresponding to `id`, `invalidId` is raised.

Example: *This primitive is more "technical", it allows a service or a peer code to know the provider of a service or its own identifier.*

[7] Subsequent invocations to `newPeer()` returns the same `peerId`.

[8] See `FaultHandler` command in Subsection 2.3.2.

Peer Management:
peerId `newPeer`(credentials)
peerId `getPeerId`(id?)
peerId[] `getPeers`(groupId)

Group Management:
groupId `createGroup`(groupDescription)
groupId[] `getGroups`(groupDescription?)
groupDescription `getGroupDescription`(groupId)
void `joinGroup`(groupId, credentials)
void `leaveGroup`(groupId)
groupId[] `getIncludingGroups`()
groupId `getPublishingGroup`(id?)

Service Management:
<groupServiceId, peerServiceId> `publish`(groupId, serviceContract)
void `unpublish`(peerServiceId)
<groupId, groupServiceId, peerServiceId>[] `getServices`(groupId?, peerId?, serviceContract?, maxResults?, credentials)
serviceContract `getServiceContract`(id)
sessionId `startSession`(serviceId)

Message Handling:
output? `invoke`(entityId, operationName, input?)
<callerId, input?> `receiveMessage`(groupId?, operationName)
void `reply`(callerId, operationName, output?, faultName?)

Event Handling:
void `subscribe`(eventName?, groupId?)
void `unsubscribe`(eventName?, groupId?)
void `event`(groupId?, eventName, input?)
<callerId, input?> `receiveEvent`(groupId?, eventName)

Figure 2.3: SMEPP Primitives.

GetPeers
`peerId[] getPeers(groupId) throws exception invalidGroupId, callerNotInGroup`
`getPeers()` returns the list of peer members of the `groupId` group. If no `groupId` group can be found, the primitive raises an `invalidGroupId`. As its name implies, `callerNotInGroup` is raised when the caller is not a member of `groupId`.

Example: *This primitive could be notably used to implement a network browser in the telecare application. It provides a list of member identifiers, which in turn can be used to browse services.*

Group management

A SMEPP peer can create, join, leave and discover groups. There are primitives for each of those functionalities. Note that access to groups is restricted, a peer needs to have the right credentials to become a group member. Furthermore, a peer may join several groups. It is important to point out that group creation, joining and leaving can only be managed by peer code (i.e. not by service code). That is why each of the corresponding primitive raises an `invalidCall` exception when called by a service code.

CreateGroup
`groupId createGroup(groupDescription) throws exception invalidCall`
To create a new group, a peer calls `createGroup()`. The `groupDescription` parameter has to contain, at least, the name of the group. Optionally, the creator can add information such as a textual group description. Remember that each group has an associated key to restrict access

to it[9]. Furthermore, the service model allows several groups to have the same name. They are only identified by the `groupId` returned by the middleware.

Example: *In the telecare application, this primitive would permit a telecare user to create a group for all his/her family and friends. Then, they could communicate with each other in a secure fashion.*

JoinGroup

`void joinGroup(groupId, credentials) throws exception accessDenied, invalidGroupId, invalidCall`

Peers use this primitive to become member of a `groupId` group. If the `credentials` actually authenticate the peer (it matches the security level of the group) and if the group exists, the access is granted. Otherwise, corresponding exceptions are raised. Note that several calls to join the same group do not raise an exception.

Example: *In the same vein as the previous primitive, this primitive would be used to allow family members or friends to join the group created by a telecared user.*

LeaveGroup

`void leaveGroup(groupId) throws exception invalidGroupId, peerNotInGroup, invalidCall`

Peers use the `leaveGroup()` primitive to exit a `groupId` group. If the peer published services in this group, all of them are automatically removed from it by the middleware.

Example: *When a relative or a friend does not want to take part of the user's undertaking anymore, s/he can leave the group. Then, s/he does not provide any service to other users anymore.*

GetGroups

`groupId[] getGroups(groupDescription?)`

This primitive is used by peers and services to discover groups available in the SMEPP application. It returns a list of group identifiers. The parameter can be used to retrieve only groups matching certain characteristics (e.g. group name, security level, etc.).

Example: *Still in the context of the telecare application, let's say there exist groups for pregnant women, diabetic persons, renal patients etc. Those groups serve to exchange information between patients in the same condition. The primitive may be used by a new user who wants to discover groups of his/her interest (specifying key words, group name etc.).*

GetGroupDescription

`groupDescription getGroupDescription(groupId) throws exception invalidGroupId`

Peers call this primitive to get more information about a group. The `groupDescription` object contains the information the group creator has provided at the creation time.

Example: *Following the previous example, this primitive may be used to get additional information on a particular group.*

GetIncludingGroups

`groupId[] getIncludingGroups() throws exception invalidCall`

`getIncludingGroups()` returns an array containing the group identifiers of the groups the caller peer belongs to.

Example: *Any user of the telecare application could call this primitive to know in which groups s/he is currently active.*

GetPublishingGroup

`groupId getPublishingGroup(id?) throws exception invalidId, invalidCall`

where `id` is either `groupServiceId`, `peerServiceId` or `sessionId`

The behaviour of this primitive depends on whether it is called by a service or by a peer. Called

[9] In security level 1.

by a service, without parameter, the primitive returns the group in which the (caller) service is published. Called by a peer or a service, `id` specified, the primitive returns the group in which the service corresponding to `id` is published. The primitive raises an `invalidCall` exception when no `id` is provided and the caller is a peer.

Example: *This primitive provides a mean for a service to know in which group it has been published.*

Service management

Services are published by peers into groups by using the `publish()` primitive. Once a service is published, it is visible both as a "peer service" and as a "group service". As explained in Subsection 2.1.1, callers use peer services when they want to interact directly with a particular peer. They use group services when they want to invoke "blindly" a service, this means without knowing the actual provider. Moreover, the service's visibility inherits from the group in which it is published. Thus, a peer can only discover services published in groups to which it has access.

The SMEPP service model specifies two kinds of services: *state-less* and *state-full*. State-less services do not keep track of the past interactions with clients, while state-full ones do. For instance, a radiation monitoring service can be modelled by a state-less service, one does not need to keep track of interactions to offer this kind of information. State-full services are divided into *session-less* and *session-full* services.

- **Session-less services** are services which have only one communication channel, shared by all clients. This communication channel is active as soon as the service is published and dies with the removal of the service. For instance, a virtual blackboard where every client can write at anytime could be implemented by using a session-less service.

- **Session-full services** support multiple channels. Each client has to open a new channel (using the `startSession()` primitive) before actually interacting with the service. Clients can also share their channel with others (by giving the session identifier to other clients).

Figure 2.4 (taken from [98]) illustrates the three types of services. The left-hand side of the picture shows a state-less service, where clients interact through their own channel. The right-hand side presents three clients interacting with a session-full service. Two clients share a channel while the last one has its own channel. The bottom of the picture illustrates a session-less service, two clients interact with it, using one shared channel.

In the following, we give the five primitives which allow to manage services.

Publish

`<groupServiceId, peerServiceId> publish(grouId, serviceContract) throws exception invalidService, invalidGroupId, peerNotInGroup, invalidCall`

This primitive is used by peers to publish a service into a group (`groupId`). The `serviceContract` parameter is the service contract which notably contains the signature and the grounding of the service. The primitive returns two identifiers corresponding to the two ways a service can be invoked. `groupServiceId` stands for the identifier to invoke the service as a "group service", while `peerServiceId` is the identifier for the invocation as a "peer service". Note that if two peers publish two services with the same contract in the same group, the middleware has to return the same `groupServiceId`. Furthermore, the "republishing" of a service does not raise an exception. A call to the primitive raises `invalidService` if the contract does not correspond to a valid service. Other throwable exceptions are straightforward. `InvalidGroupId` is raised when no `groupId` can be found and `peerNotInGroup` is raised when the caller is not member of `groupId`. Remember that a service code cannot call `publish()`.

Example: *In the context of the plant monitoring application, this primitive may be used by a worker node to provide a service which allows the node to be configured remotely. For instance, this service would exhibit an operation for each editable configuration details.*

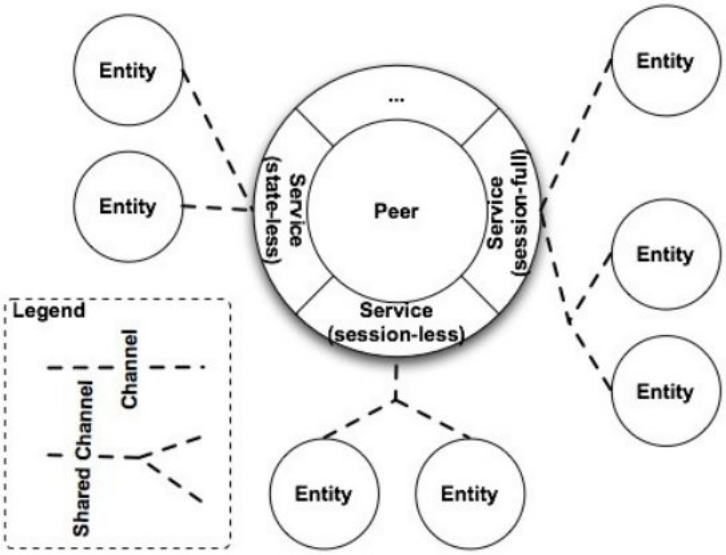


Figure 2.4: Service types.

Unpublish

`void unpublish(peerServiceId) throws exception invalidServiceId, peerNotServiceOwner, invalidCall`

`Unpublish()` is the dual of `publish()`, it removes the service from the list of discoverable services inside a group. Exceptions are raised if `peerServiceId` does not exist, if the caller is not the service owner or if the caller is a service code.

Example: *In the telecare application,* `unpublish()` *may be used by a doctor when s/he finishes her/his working day. For instance, if the doctor offers a teleconsulting service from 8:00 a.m. to 17:00 p.m., the service may be unpublished at 17:00 p.m.*

GetServices

`<groupId, groupServiceId, peerServiceId>[] getServices(groupId?, peerId?, serviceContract?, maxResult?, credentials) throws exception invalidGroupId, invalidPeerId, invalidService`

Services discovery is done by using `getServices()`. The `credentials` parameter serves to match the security level of SMEPP groups in which services are published. All the other parameters are optional, the output of the primitive is filtered to match them. The primitive raises exceptions when the specified identifiers do not exist or when the provided service contract is not valid.

Example: *Following the previous example, this primitive may be used by patients to discover which services a doctor offers.*

GetServiceContract

`serviceContract getServiceContract(id) throws exception invalidId`

where `id` is either `groupServiceId`, `peerServiceId` or `sessionId`

Entities call `getServiceContract()` to retrieve the service contract of a service corresponding to `id`.

Example: *Let's say a user has retrieved all the service identifiers corresponding to her/his doctor. By using this primitive, the user can get enough information to actually invoke the service.*

StartSession

`sessionId startSession(serviceId) throws exception invalidServiceId, accessDenied, cannotStartSession`

where `entityId` is either `groupServiceId`, `peerServiceId` of a session-full service
A communication channel, or session is created with `startSession()`. It returns an identifier of the session, which can be used by the caller to communicate with an instance of the service. Sessions can be shared among peers at the application level. Furthermore, a peer can interact with a service using multiple sessions. If the caller specifies a `groupServiceId`, the middleware has to choose an actual provider (`peerServiceId`) and then starts a session with it. The `invalidServiceId` exception is raised when no corresponding service can be found, `accessDenied` when the caller and the provider do not belong to the same group and `cannotStartSession` when the provider cannot start a new session (e.g. if the maximum number of open sessions has been reached).

Example: *This primitive may be used in the chat service of the telecare application. For instance the chat program could permit both public and private discussions. Starting a session would correspond to create a new private channel between two users.*

Message handling

In SMEPP, messages are used to invoke service operations. The model features two kinds of operation. *One-way* operations take only input parameters, while *request-response* operations return a result to the caller.

Basically, the invocation of an operation is done as follows. Firstly, the caller sends a message, containing the (possibly empty) input parameters via the `invoke()` primitive. Then, the provider calls `receiveMessage()` in order to retrieve the input parameters. Finally, if the operation is "request-response", the provider returns the result by using `reply()`. Note that the provider could call `receiveMessage()` before the caller calls `invoke()`. In that case, the provider's program would be blocked until a corresponding message can be retrieved.

The three primitives are detailed below. Regarding the caller program, if the operation type is one-way, it will be blocked until the provider makes a corresponding call to `receiveMessage()`. If the operation type is request-response, the caller is blocked until it gets the result (i.e. when the provider calls the corresponding `reply()`.

It is also important to note that the model does not allow a same entity to execute a same concurrent request-response operation of a same provider (except if it is managed by different sessions).

Invoke

`output? invoke(entityId, operationName, input?) throws exception invalidPeerId, invalidServiceId, invalidOperation, concurrentRequest, invalidInputParameter, invalid-OuputParameter, accessDenied`

where `entityId` is either `peerServiceId`, `groupServiceId`, `sessionId` or `peerId`
Entities use `invoke()` to call operations (identified by `operationName`) of a provider (identified by `entityId`). `Invoke()` imposes the following restrictions:

- If the invoked service is **state-less**, `entityId` must be either `groupServiceId` or `peerServiceId`.
- If the invoked service is **session-less**, `entityId` must be `peerServiceId`. Note `entityId` cannot be a `groupServiceId` since a state-full service[10] keeps track of past interactions and the middleware does not guarantee the same provider will be chosen between two invocations specifying the same `groupServiceId` identifier.
- If the invoked service is **session-full**, `entityId` must be `sessionId`, since session-full services can only be called through sessions.

Note that `entityId` can be a `peerId` identifier (direct interaction with the peer code). Furthermore, the model makes the assumption that operations offered directly by peers are *one-way*. In the other cases, the middleware identifies the operation type from the service contract.

Regarding exceptions, `invalidPeerId`, `invalidServiceId` and `invalidOperation` are raised when the corresponding object cannot be found. `ConcurrentRequest` is raised when the caller is

[10] Remember that session-less and session-full services are both state-full services.

already calling this operation on this provider. `AccessDenied` is raised when the two entities do not belong to the same group. Finally, `invalidInputParameter` and `invalidOuputParameter` are raised when the parameters do not match the operation signature.

Example: *In the plant monitoring application, this primitive may be used by a supervisor node to remotely configure a worker node (one-way operation). In the telecare application, it could be used by patients to do a health check-up. They could send some data (such as temperature, blood pressure, etc.) to their doctor and receive her/his opinion (request-response operation).*

ReceiveMessage

`<callerId, input?> receiveMessage(groupId?, operationName) throws exception invalidOperation, invalidGroupId, callerNotInGroup, invalidInputParameter`

where `callerId` is either `peerServiceId`, `sessionId` or `peerId`

Entities call this primitive to retrieve an invocation message for `operationName`. Service calls to `receiveMessage` require that a service offering `operationName` is published by the caller. The primitive returns the identifier of the caller (`callerId`) and the input parameters. The optional `groupId` parameter restricts the reception of the message to the group identified by the respective `groupId`.

Example: *This primitive may be used by a service (running at the doctor side) waiting for some patient's data in order to give a diagnosis.*

Reply

`void reply(callerId, operationName, output?, faultName?) throws exception invalidPeerId, invalidPeerServiceId, invalidOperation, missingReceiveMessage`

where `callerId` is either `peerServiceId`, `sessionId` or `peerId`

Provider of request-response operations use `reply()` to give the result of an operation (`operationName`) to its caller (`callerId`). Note that the model requires that a corresponding message has been previously received via a call to `receiveMessage()`, otherwise `missingReceiveMessage` is raised. The primitive can also be used to signal an erroneous behaviour of the operation to the caller. In such a case, the `output` parameter stands for the data associated with the fault.

Example: *In the same vein as the previous example, this primitive may be used by the doctor to send back her/his diagnosis to a patient.*

Event handling

Basically, event-based communication differs from message-based communication in two ways. Firstly, the generator of the event does not block until someone receives it. Secondly, more than one client can receive the same event. Entities have to subscribe (`subscribe()` primitive) to the events of their interest in order to receive them. Once subscribed, entities can actually receive an event via the `receiveMessage()` primitive. On the provider side, raising an event is done by using the `event()` primitive.

Event

`void event(groupId?, eventName, input?) throws exception invalidGroupId, callerNotInGroup, invalidEvent`

This primitive serves to raise an `eventName` event. If `groupId` is specified, the event is published in this group. Otherwise, if the primitive is called by a service code, it is raised in the group in which the service is published. If the caller is a peer and the group is not specified, the event is published in all groups to which the peer belongs. The caller can also attach additional data with the `input` parameter. It is important to highlight that the service model does not specify any lifetime for events.

Exceptions are similar to `receiveMessage()`. However, note that a service call to `event()` implies a previous publication of a service contract that defines `eventName`, otherwise `invalidEvent`

is raised.

Example: *In the context of the plant monitoring application, raising an event may be used to signal a radiation alert. For instance, when a worker node detects an abnormal radiation level, it raises an event called "radiation level exceeded" with the measured level as associated data.*

ReceiveEvent
`<callerId, input?> receiveEvent(groupId?, eventName) throws exception invalidGroupId, callerNotInGroup, invalidInputParameter`
where `callerId` is either `peerServiceId`,`sessionId` or `peerId`
To retrieve an event, entities call `receiveEvent()`. This primitive is similar to `receiveMessage()`. But in this case, entities have to be subscribed to `eventName` before being able to receive such an event.

Example: *Likewise in the previous example, this primitive may be used at the supervisor node side, to receive an event signalling a radiation alert.*

Subscribe
`void subscribe(eventName?, groupId?) throws exception invalidGroupId, callerNotInGroup`
To register as event listeners, entities call `subscribe()`. They can subscribe to:

- `eventName` events raised in a `groupId` group,
- `eventName` events raised in all groups (to which the caller belongs),
- all events raised in a `groupId` group, or
- all events raised in all groups to which the caller belongs.

Exceptions raised by `subscribe()` are similar to the primitive seen before.

Example: *In the context of the telecare application, this primitive may be used by relatives and friends of a patient to signal to the middleware they want to receive events concerning the patient. For instance, an event could be an emergency alert or a reminder spread in the patient's family/friend group.*

Unsubscribe
`void unsubscribe(eventName?, groupId?) throws exception invalidGroupId, callerNotInGroup, notSubscribed`
This primitive is the dual of `subscribe()`, it cancels previous subscriptions. Note that the two primitives do not have to match exactly. For instance, `unsubscribe(eventName, groupId)` matches a previous `subscribe(eventName)`. `Unsubscribe()` raises `notSubscribed` when there are no matching previous subscriptions. Other exceptions are similar to the primitives seen before.

Example: *Still in the telecare application, this primitive may be used by users who do not want to receive emergency alerts anymore (because they are abroad, for example).*

2.3.2 SMoL details

The SMEPP Modelling Language (SMoL for short) is used to orchestrate the primitives in order to define the behaviour of peers and services. The language is similar to (and is inspired by) the Business Process Execution Language (BPEL [78, 79]). Since the language takes a lot of concepts from BPEL, it is useful to present this reference language. The first part of this subsection briefly describes the main ideas of BPEL. Then, we detail SMoL by giving its constructions and, informally, their semantics. Further in this chapter, Subsection 2.3.3 proposes an abstract model of the language.

Business Process Execution Language.

The second version of BPEL has been specified by the Organization for the Advancement of Structured Information Standards (OASIS), a consortium that drives the development of open standards for the global information society. This version is the result of a long process involving the combination of several "programming in the large languages" such as WSFL (IBM) and XLANG (Microsoft).

The definition of the language takes place in the field of Web Services which bring interoperability between heterogeneous applications through web standards. The interaction between business processes is often complex and long-running, involving sequences of message exchanges between several parties. To define such business interactions, one needs a formal description of the message exchange protocols used by the applications. An "Abstract Process" can describe observable behaviour of all parties involved in the interaction. On the one hand, this allows businesses not to reveal all their internal decision making and data management. And, on the other hand, it provides the freedom of changing private aspects of the implementation without affecting the observable behaviour.

BPEL defines two kinds of processes, Abstract and Executable. An Abstract Process is a partially specified process, it may be used to describe a process template. Then, it would capture essential process logic while excluding implementation details. An Executable Process is fully specified and thus can be executed. Both Abstract and Executable Processes share the same constructs, they thus have the same expressive power. In concrete terms, the language offers constructions for the specification of Abstract and Executable processes. For instance, the language provides constructions similar to what can be found in classic programming languages, e.g. sequences, repeated executions (loops), conditional branches etc. It also features parallel execution and exception handling. BPEL code is written using XML. This allows processes to be human-readable while being easily processed by XML tools and other parsers. The BPEL process is basically an expression of an algorithm of which each step is called an activity. Those activities are similar to the SMEPP primitives (actually, the primitives are inspired by the BPEL activities). For instance, one can invoke an operation on a web service via <*invoke*>, wait for a message (<*receive*>), etc. Activities are combined using "structured activities" to create more complex algorithms. One can define an ordered sequence of steps (<*sequence*>), execute one of several alternative paths (<*pick*>), execute several steps in parallel (<*flow*>) etc. The language allows to recursively combine the constructions.

The SMEPP Modelling Language (SMoL) is a BPEL-like language. Many constructions available in SMoL are taken from BPEL. SMoL programs are also written by using XML. Furthermore, the purpose of SMoL is quite similar to BPEL's one. Indeed, SMoL allows to model the behaviour of peer and service codes as BPEL models business processes. It is then possible to analyse formally the interactions between peers and/or services (as BPEL allows the analysis of interactions between business processes). However, SMoL was designed to simplify BPEL in the context of the SMEPP project. Indeed, the semantics of BPEL is quite complex and, thus, its analysis is very time-consuming. Moreover, SMEPP, being targeted to the field of EP2P, does not require several BPEL concepts [15]. The removed concepts (or constructs) are[11]:

- compensation which allows to specify an activity that is used to undo one of the steps that have already been completed,
- synchronisation links, which are used to declare control dependencies between concurrent activities,
- forEach which is similar to the `for` statement in classic programming language, however it allows parallel execution of its body,
- isolated scope which, basically, provides control of concurrent access to shared resources. It is similar to the concept of "serialisation" in database transaction,

[11] This list describes some BPEL constructs in very few words, please refer to [79] for more information.

- partner links which define the different parties involved in a process (e.g. customer, provider, shipping provider, etc.),
- message properties which provide a way of naming and representing (business or infrastructure) protocol relevant data,
- correlation sets which are used to "tag" conversations involving several parties. They allow to match messages with business process instances for which they are intended.

The rest of this subsection describes in details SMoL. Note that all of the following commands are very similar to BPEL's ones.

SMEPP Modelling Language

The constructions (or commands) of the language are typically subdivided in two categories. A basic command is either a primitive call (see Figure 2.3 for a summary of the primitives) or a call to an atomic statement. A structured command orchestrates basic commands. They can be used recursively to define more complex behaviours. Note that a lightweight formal semantics of the language can be found in [15]. This provides a way to analyse formally the peer and service behaviour.

Basic commands. Basic commands are either a command listed below or a SMEPP primitive. The following commands are somewhat similar to the SMEPP primitives. However, the basic commands only define local behaviours. In other words, a basic command affects only its local program, not other peers or services.

Some parameters used in those commands are specified by using the XML Path Language (XPath [110]). Basically, XPath is a language for selecting nodes from an XML document. In SMoL, this functionality is used to retrieve and store values. Futhermore, it can also be used to compute boolean, integer or string values.

Empty
`void empty()`
A call to `empty()` is equivalent to "no-op". It serves in case one wants an execution branch to do nothing (e.g. ignore a fault in a `FaultHandler`, see below).

Wait
`void wait(for?, until?, repeatEvery?)`
A program calls `wait` to suspend its execution either for a certain time (`for`) or until a certain moment (`until`). In case the command is used inside an `InformationHandler` (see below), programmers may use the `repeatEvery` parameter to trigger a certain branch repeatedly after each specified period of time. It is important to note that only one parameter can be used at a time. The syntax is defined by XPath. For instance, a program executing `wait(2009-09-15)` will wait until the 15^th of September 2009 and a program executing `wait(P3DT10H)` will wait during three days and ten hours.

Throw
`void throw(faultName, faultVariable?)`
`Throw` is used to raise a fault inside a program. The first parameter defines the name associated with the fault (e.g. "connection lost"). The second one defines the data associated with the fault, this data is used for debugging purpose.

Catch/CatchAll
`faultVariable? catch(faultName)` ***and*** `<faultName, faultVariable> catchAll()`
Those commands serve to "identify" faults raised in programs. They can only be used inside `faultHandler`s (see below). The parameter of a `catch` is the name of the fault to be caught and the command returns the data associated with it. `CatchAll` catches all faults. Once a fault is caught, it outputs its name and associated data.

Exit

```
void exit()
```

`Exit` simply terminates the execution of the program (all running commands are stopped), it is similar to `exit(0)` in the C programming language.

Assign

```
Assign
        Copy
                  from
                  to
        End Copy
        ...
        Copy
                  from
                  to
        End Copy
End Assign
```

Programmers use this command to assign values to variables. It is similar to any assignment in a classic programming language except that, in this case, one can do several assignments at once. XPath can be used to define the `from` and `to` parameters.

Structured commands. Structured commands orchestrate SMEPP primitives and SMoL (basic or not) commands. This part of the text intends to give an informal description of each command. In the following, `command` refers to any SMoL command.

Sequence

```
Sequence
        command
        ...
        command
End Sequence
```

Basically, a `sequence` provides a way to execute a set of commands in lexical order. In a classic programming language, one would have used semi-colons between each `command`.

Flow

```
Flow
      command
      ...
      command
End Flow
```

This command allows several commands to be executed at the same time, that is concurrently. The execution of the `Flow` terminates when all of its child commands have finished their execution. This construction is somewhat similar to the notion of "thread" in the C or the Java programming language.

While

```
While boolCond
          command
End While
```

`While` has the same semantics in SMoL than in other programming languages. It executes the `command` as long as `boolCond` is satisfied and the condition is evaluated before each cycle. The condition is assumed to be defined with XPath.

RepeatUntil

```
RepeatUntil boolCond
                command
End RepeatUntil
```

`RepeatUntil` is similar to `While` but it evaluates `boolCond` *after* each cycle. Thus, in any case `command` is always executed at least once.

If-then-else

```
If boolCond
        command
Else
        command
End If
```

This command implements simply the classic conditional control-flow. It executes either the first `command` if `boolCond` is evaluated to `true` or the second one if it is evaluated to `false`. Note that the `Else` branch is optional.

Pick

```
Pick
       <callerId, input?>   = receiveMessage(groupId?, operationName)
                            command
       ...
       <callerId, input?>   = receiveEvent(groupId?, eventName)
                            command
       ...
       wait(for?, until?)
                            command
       ...
End Pick
```

The `Pick` command introduces non-deterministic choices in SMoL. The branch to be executed is chosen depending on the time elapsed and the messages/events received by the program. Thus, the execution of `pick` amounts to wait for an invocation message to be received, an event to be raised or a timer to go off. The first event or message received selects the `command` to be executed. However, if a timer terminates before, its branch is executed. It is important to insist that in any case, only one `command` is executed.

InformationHandler

```
InformationHandler
        command
        <callerId, input?>              = receiveMessage(groupId?, operationName)
                                        command
        ...
        <callerId, input?>              = receiveEvent(groupId?, eventName)
                                        command
        ...
        wait(for?, until?, repeatEvery?)
                                        command
        ...
End InformationHandler
```

This command executes a main `command`, while in parallel, it receives and processes messages/events as well as alarms. The main `command` is placed in first position inside the `InformationHandler`. The command is similar to a `Pick`, it specifies a set of messages, events and alarms, each of them associated with a `command`. However, the execution of an `InformationHandler` terminates when its main `command` finishes. In addition, it can receive several messages/events during its lifetime. Furthermore, the `repeatEvery` parameter of the `wait` command can be set to execute repeatedly a `command`. It is important to note that when the `InformationHandler` finishes, the running `command`s (inside `receive` and `wait` branches) must be allowed to terminate.

FaultHandler

```
FaultHandler
        command
                faultVariable1?  = catch(faultName1)
                command
        ...
                faultVariablen?  = catch(faultNamen)
                command
                <faultName, faultVariable?> = catchAll()
                command
End FaultHandler
```

The `FaultHandler` serves to process faults which may be raised inside a program (i.e. the main `command` associated with the `FaultHandler`). When a fault is raised, the execution of the main `command` is stopped and the list of `catch`es is processed until one of the clauses matches. Then, the `command` associated with the matching `catch` is executed. If no matching `catch` can be found, the `command` associated with the `catchAll` branch is executed. Note that a `FaultHandler` without a `catchAll` branch is assumed to have an implicit `catchAll` branch which forwards every fault to the outer `FaultHandler`. In case it is the outermost `FaultHandler`, the fault is forwarded to the environment (e.g. to the human user). The structure and the purpose of this construct is similar to a try/catch block in Java.

2.3.3 Abstract model

In order to focus on the coordination challenges implied by the service model presented above, we need to define a more formal model for it. This model abstracts many details which are not relevant from the coordination point of view such as input/output data types and optional parameters.

The model we propose is based on [13], which defines a calculus for SMEPP primitives. We extend it with group related security and a comprehensive calculus for SMoL, while trying to make some concepts easier to understand. This is achieved thanks to the following modifications. We represent the entities using always the same notation, while in [13] the notation differs depending whether the peer is authenticated or not. Moreover, the modelling of event

publications and subscriptions are simplified by using two sets belonging to a group instead of using the so-called *polling-context*. Finally, we present the rules in a more readable format by clearly separating different classes of conditions (on groups, on peers, etc.).

Let us first introduce some notations. Let $\mathcal{P}$ be the set of peer identifiers, $\mathcal{S}$ the set of service identifiers, $\mathcal{E}$ the set of event names and $\mathcal{G}$ the set of group identifiers. We assume that $\mathcal{G}$ contains a special group "0" which gathers all the peers of a SMEPP application. In the same vein, let $\mathcal{I}$ be a set which contains every possible input for every event and $\mathcal{C}$ be the set which contains every service contract.

Thanks to these notations, each system entity is identified by a middleware uniform resource locator ($\mathcal{MURL}$), $g.p.s \in \mathcal{G} \times \mathcal{P} \times \mathcal{S}$. Moreover a service is denoted by a triple, <*s,cs,p*>, where s $\in \mathcal{S}$, cs $\in \mathcal{C}$ (the set of service contracts) and p $\in \mathcal{P}$. We assume the operation signatures to be included in the contracts.

We model a group g protected by a password *pwd* by a 4-tuple : $\langle P, Sr, Sb, Pb \rangle_{g,pwd}$, where:

- $P \subseteq \mathcal{P}$ represents the peers members of g,
- $Sr \subseteq \mathcal{S}$ represents the services published in g,
- $Sb \subseteq \mathcal{MURL} \times \mathcal{E}$ represents the subscription of an entity $g.p.s$ to an event and
- $Pb \subseteq \mathcal{MURL} \times \mathcal{MURL} \times \mathcal{E} \times \mathcal{I}$ represents the set of event publications. Note that *(g.p.s,g'.p'.s',n,in)* is the instance of an event n raised by $g.p.s$ to be received by *g'.p'.s'* containing the input *in*.

As described below, a peer executes its peer code which may offer services. Both peer and service code are modelled by: $[A]_{g.p.s,pwdlist}$, where $g.p.s$ is the entity's $\mathcal{MURL}$ and *pwdlist* is the password list known by the peer. Note that if the program is not yet identified by the SMEPP application, $g.p.s$ is equal to $\varnothing$.

Let Π be the set of agents or programs running in the considered SMEPP application. In general, a program or agent code A meets the syntax given in Figure 2.5. There, "0" represents the empty program and "b" is a `catch()` command. The operators are the classical ones: "." represents the prefix, "$\|$" is the parallel composition, "$\oplus$" is the guarded non-deterministic choice. Also, as in [98], we use the "@" operator to model exception handling. The "Es" element is a sequence of guarded (by catch constructions) non-deterministic choices. Please note that $[A]_{g.p.s}$ is written as a shorthand for $[A@[]]_{g.p.s}$; this equivalence helps to completely define the exception handling rules. We finally pose Γ as the set of groups available in the SMEPP application.

The first element (`A`) of Figure 2.5 represents a SMEPP instruction which can be either a SMoL command or a SMEPP primitive. A SMoL program is a SMEPP statement. SMoL has two kinds of commands: basic commands and structured ones. There are four basic commands:

- `Empty` is the empty program, modelled as 0,
- `Exit()` terminates the execution of a program,
- `Wait(x)` blocks the execution of the code during "x" time intervals or until time "x",
- `Throw(e,in)` raises an exception "e" with the parameters "in".

Structured commands contain SMEPP instructions and/or logical expressions. The following lines explain the reduction rules referring to the structured commands.

- `Assign(CFT)`, where `CFT` is a list of `(from x to y)` constructs, reduces to the execution of the first element of `CFT` to `y=x`, executed in parallel with the application of the rule on the rest of the list (using the operation " $\|$"),
- `Sequence(As)` reduces to the execution of the first element of `As` (a SMEPP instruction) prefixing the application of the rule on the rest of the list,
- `Flow(As)` reduces to the parallel execution of the first element of `As` and the result of the application of the rule on the rest of the list,

```
A                              ::=  Empty()  | Exit()  | Wait(x)  |
                                    Throw(e,in)  | Assign(CFT)  |
                                    Sequence(As)  | Flow(As)  | While(BL,A)
                                    | RepeatUntil(BL,A)  | If(BL,A1,A2)  |
                                    Pick(Bs)  | InformationHandler(A,Is)  |
                                    FaultHandler(A,Es)  | P
Empty()                        ::=  0
Exit()                         ::=  termination of surrounding program
                                    execution
Wait(x)                        ::=  delays the execution for ``x'' time
                                    intervals or until the time ``x''
Throw(e,in)                    ::=  raise(e,in)
Assign((from x to y):CFT)      ::=  y=x ‖ Assign(CFT)
CFT                            ::=  []  | (from x to y):CFT
Sequence(A:As)                 ::=  A.Sequence(As)
Flow(A:As)                     ::=  A ‖ Flow(As)
As                             ::=  []  | A:As
While(BL,A)                    ::=  if BL then A.While(BL,A)
RepeatUntil(BL,A)              ::=  A.While(BL,A)
If(BL,A1,A2)                   ::=  if BL then A1 else A2
BL                             ::=   logical expression
Pick(BR(A):Bs)                 ::=  BR(A) ⊕ Pick(Bs)
InformationHandler(A,Bs)       ::=  (x=true).A.(x=false) ‖ IHB(Is)
IHB(BR(A):Bs)                  ::=  While(x,BR(A)) ‖ IHB(Bs)
Bs                             ::=  []  | BR(A):Bs
BR(A)                          ::=  receiveEvent().A  | receiveMessage().A  |
                                    wait().A
FaultHandler(A,Es)             ::=  A@Es
Es                             ::=  []  | (b.A):Es
b                              ::=  catch(e,in)
P                              ::=  x = newPeer(pwdlist)  | createGroup(g)
                                    | joinGroup(g,pwd)  | leaveGroup(g)  |
                                    x = publish(g,cs)  | unpublish(m)  | x
                                    = getServices(m,cs)  | startSession(m)
                                    | subscribe(g,n)  | unsubscribe(g,n)  |
                                    event(g,n,in)  | ⟨x,in⟩ = receiveEvent(g,n)
                                    | x = invoke(m,n,in)  | reply(m,n,out)
```

Figure 2.5: Program syntax.

- `While(BL,A)` reduces to the evaluation of the condition `BL` prefixing the execution of `A` which is followed by the application of `While(BL,A)`, recursively.
- `RepeatUntil(BL,A)` reduces to the execution of `A` prefixing the application of `While(BL,A)`,
- `If(BL,A1,A2)` reduces to the well known *if then else* statement, where `BL` is the boolean condition, `A1` is the *then* branch, and `A2` is the *else* one.
- `Pick(Bs)` reduces to a non-deterministic choice between the first element of `Bs` and the result of the rule application on the rest of the list, where `BS` is composed of `BR(A)`s,
- `InformationHandler(A,Bs)` reduces to the parallel composition of the branches (`BR(A)`) contained in `Bs` and the main command `A`, surrounded by a boolean flag modelling `A`'s state of execution,
- `BR(A)` models the possibles branches of `Pick` and `InformationHandler` commands. Such a branch reduces to the execution of a primitive (which must be one of the list) followed

by its attached command `A`,

- `FaultHandler(A,Es)` is modelled using the "@" operator, the semantics of "@" is detailed later in the exception handling rules. `Es` is a list of catch clauses.

As in [13] and [98], we have simplified the input and output of the primitives for the sake of simplicity and to better highlight the coordination challenges. The signature considered for SMEPP primitives is defined as g for `groupId`, cs for `serviceContract`, m for $\mathcal{MURL}$, n for operation, *in* for input, *out* for output, *pwdlist* for the credentials, *pwd* for one group key and e for an exception name.

In order to ease the reading of the rules, we give the semantics of the primitives using rules of the form:

$$(ex\_rules)\quad \frac{\begin{array}{c}\textit{Various provisos}\\ \textit{Provisos on } \Pi\\ \textit{Provisos on } \Gamma\end{array}}{\begin{array}{c}\textit{State of } \Pi\\ \textit{State of } \Gamma\end{array}}$$

Remember that Π is the set of running agents/programs and Γ is the set of active groups. We apply the rules in a top-down way, until the program codes in Π are terminated (i.e. $\Pi = \varnothing$) or locked (i.e. $\Pi \neq \varnothing$ and no more applicable rules).

Peer management

$$(newPeer)\quad \frac{\begin{array}{c}(0, ApKey) \in pwdlist\\ \Pi, [(x = newPeer(pwdlist).A)@Es]_{\varnothing,pwdlist}\\ \Gamma \cup \langle P, \varnothing, \varnothing, \varnothing\rangle_{0,ApKey}\end{array}}{\begin{array}{c}\Pi, [(A[p/x])@Es]_{0.p.0,pwdlist}\\ \Gamma \cup \langle P \cup \{0.p.0\}, \varnothing, \varnothing, \varnothing\rangle_{0,ApKey}\end{array}}$$

The above rule represents the semantics of the `newPeer()` primitive. The first proviso line requires a peer to have the couple $(0, ApKey)$ in order to access the universal group (0), with $ApKey$, the SMEPP application password. The second line requires the existence of a non yet authenticated peer (labelled by $\varnothing$) which is about to execute `newPeer()`. One can also see that the future peer has a list of password *pwdlist*. The third line models the state of the groups in the SMEPP application. The universal group (0) is highlighted. By definition this is a special group which does not contain any service or event. Note that we pose that $g.p.s \neq \varnothing$ for each $g.p.s \in \mathcal{MURL}$. In this way, a program cannot call other primitives before calling `newPeer()`.

As a result, the program "A" gets labelled with $0.p.0$ and not $g.p.s$ (which could be the MURL of a service). This ensures that the program is considered as a peer code instead of a service code (remember a service cannot call `newPeer()`). In the rest of A's code, the result of `newPeer()` replaces the free occurrences of x. This is modelled by the first line of the conclusion. In the second line, the peer is added into the set of peers of the universal group: $P \cup \{0.p.0\}$.

Group management

This paragraph gives the semantics of group management primitives. We detail three primitives: `createGroup()`, `joinGroup()` and `leaveGroup()`.

$$(createGroup)\quad \frac{\begin{array}{c}(g, pwd) \in pwdlist\\ \Pi, [(x = createGroup(g).A)@Es]_{0.p.0,pwdlist}\\ \Gamma; \langle *, *, *, *\rangle_{g,pwd} \notin \Gamma\end{array}}{\begin{array}{c}\Pi, [A[g/x]@Es]_{0.p.0,pwdlist}\\ \Gamma \cup \langle \{0.p.0\}, \varnothing, \varnothing, \varnothing\rangle_{g,pwd}\end{array}}$$

This rule gives the semantics of the `createGroup()` primitive. The first line enforces the caller peer to have the password (*pwd*) associated with the group it wants to join (g). The second line shows that the set of executing programs contains a peer (identified by $0.p.0$) which is about to execute `createGroup()`. This peer has a list of password *pwdlist*. The third line requires that the SMEPP application does not yet contain a group labelled by g and protected by *pwd*.

As shown in the first line of the conclusion, the peer code continues by replacing every free occurrence of x by g. The last line models the addition of the newly created group g to Γ, the set of groups.

$$\textit{(joinGroup)} \quad \frac{\begin{array}{c}(g, pwd) \in pwdlist \\ \Pi, [(joinGroup(g, pwd).A)@Es]_{0.p.0,pwdlist} \\ \Gamma \cup \langle P, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [A@Es]_{0.p.0,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}$$

The rule modelling `joinGroup()` is similar to `createGroup()`, but this time the group g protected by *pwd* must already be in Γ. As a consequence of the rule, the peer *0.p.0* is added into the peers set of g.

$$\textit{(leaveGroup)} \quad \frac{\begin{array}{c}\Pi, [(leaveGroup(g).A)@Es]_{0.p.0,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [A@Es]_{0.p.0,pwdlist} \\ \Gamma \cup \langle P, Sr \backslash \{g.p.*\}, Sb \backslash \{(g.p.0, *)\}, Pb \backslash \{(g.p.0, *, *, *)\}\rangle_{g,pwd}\end{array}}$$

This rule gives the semantics of `leaveGroup()`. A peer *0.p.0*, member of the group g (protected by *pwd*), is about to execute `leaveGroup()`, as shown in the first line. The second line gives the condition on the groups: the group g protected by *pwd* must exist in the SMEPP application.

The result of the rule shows (in the last line) that the peer is removed from g's P set. The services and events related to the leaving peer must also be removed, this is done by removing the couples $(g.p.0, *)$ from Sb and all the couples $(g.p.0, *, *, *)$ from Pb where we consider $*$ as the traditional wildcard, which matches every identifier.

Service management

This paragraph gives the semantics of the service related primitives. `Publish()`, `unpublish()`, `getService()` and `startSession()` are modelled in the following four rules.

$$\textit{(publish)} \quad \frac{\begin{array}{c}(s, cs, p) \notin Sr \\ \Pi, [(publish(g, cs).A)@Es]_{0.p.0,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [A@Es]_{0.p.0,pwdlist}, [program(cs, p)]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr \cup \{(s, cs, p)\}, Sb, Pb\rangle_{g,pwd}\end{array}}$$

The rule above models `publish()`. The first line asserts the specified service is not published in g (i.e. it is not in g's Sr set). The next line shows the peer is about to execute the primitive. Note that it is labelled by $0.p.0$ to enforce the primitive to be executed by a peer code (and not a service). The two primitive arguments are the group in which the service has to be published and the service contract (cs), respectively. The third line simply shows that the group g protected by *pwd* exists.

As a result, the first line of the conclusion states that the primitive has been executed and a new program is part of the SMEPP application: an agent $[program(cs, p)]_{g.p.s,pwdlist}$ is added to Π in order to represent the running service. The last line states that the service is added to g's Sr set.

$$\textit{(unpublish)}\quad \frac{\begin{array}{c}\Pi, [(unpublish(g.p.s).A)@Es]_{0.p.0,pwdlist}, [program(cs,p)]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr \cup \{(s,cs,p)\}, Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [A@Es]_{0.p.0,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr, Sb \backslash \{(g.p.s,*)\}, Pb \backslash \{(g.p.s,*,*,*)\}\rangle_{g,pwd}\end{array}}$$

This rule models `unpublish()`. The first line shows that $0.p.0$ is about to execute the primitive in order to remove $g.p.s$. Another agent is present, representing the running service to be removed. The second line models the fact that $g.p.s$ is published. It is an element of g's Sr set.

The first line of the conclusion requires that the execution of the service is terminated, while the second requires that the service is removed from Sr and that all event elements related to $g.p.s$ are removed from Sb and Pb, as done in `leaveGroup()`.

$$\textit{(getServices)}\quad \frac{\begin{array}{c}g''.p''.s'' \leq g'.p'.* \\ \Pi, [(x = getServices(g'.p'.*, cs).A)@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P \cup (0.p.0), Sr \cup (s'', cs, p''), Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [(A[g''.p''.s''/x])@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P \cup (0.p.0), Sr \cup (s'', cs, p''), Sb, Pb\rangle_{g,pwd}\end{array}}$$

The above rule provides the semantics of service discovery (`getService()` primitive). An entity looks for a service identified by $g'.p'.*$ (where g' and/or p' can be left unspecified, using $*$) providing the service contract[12] cs. In order to use the wild card notation, we define an ordered relationship in $\mathcal{MURL}$ extended by adding $*$ to $\mathcal{G}$, $\mathcal{P}$ and $\mathcal{S}$, defined by $g.p.s \leq g'.p'.s'$ if and only if $g = g'$ or $g' = *$, and $p = p'$ or $p' = *$, and $s = s'$ or $s' = *$. Note that without loss of generality, we simplify the primitive so that it returns only one service identifier, instead of collecting service identifiers.

The first line of the rule requires that the service to be returned matches the specified argument, while the second shows the entity $g.p.s$ is part of the SMEPP application (note this is the first time we present a primitive which a service can call). The third line models the set of groups where the group g contains a matching service (described as the triple (s'', cs, p'').

The first line of the conclusion states that the peer code execution continues by replacing x by the primitive result. The last line states that the conditions on Γ are not affected by the primitive.

$$\textit{(startSession)}\quad \frac{\begin{array}{c}\Pi, [(startSession(g.p'.s').A)@Es]_{g.p.s,pwdlist}, [Program(cs,p')]_{g.p'.s',pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0, 0.p'.0\}, Sr \cup \{s', cs, p'\}, Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [A@Es]_{g.p.s,pwdlist}, [Program(cs,p')]_{g.p'.s',pwdlist}, [Session(g.p'.s',n)]_{g.p'.s'n,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0, 0.p'.0\}, Sr \cup \{s', cs, p'\}, Sb, Pb\rangle_{g,pwd}\end{array}}$$

This rule models the session creation, i.e. `startSession()`. The first line shows that an entity is about to execute the primitive on the (assumed) session-full service $g.p'.s'$. This service is also modelled in the first line as a running entity. The second line shows that the invoker and the provider are in the same group, g. Furthermore, it enforces the service $g.p'.s'$ to be published in g.

The first line of the conclusion shows that a new entity is added to Π, this entity represents the newly created session. The session is identified by the service identifier ($g.p'.s'$) and a session instance number (n). The second line remains the same.

Note that when an entity invokes an operation exhibited by a session-full service, it must specify the session identifier, i.e. the session instance number appended to the service identifier (e.g. $g.p'.s'n$).

Message management

This paragraph gives the semantics of message related primitives. The two first rules give the semantics of the `invoke()` and `receiveMessage()` primitives according to which kind of

[12] We assume, for the sake of simplicity, that the service contract is complete, i.e. it is not a template contract.

operation is called (one-way/asynchronous or request-response/synchronous). The last one describes `reply()`.

$$\textit{(invoke(sync))}\quad \frac{\begin{array}{c} n \in cs \\ \Pi, [(x = invoke(g.p'.s', n, in).A)@Es]_{g.p.s,pwdlist}, \\ [(\langle x, inp\rangle = receiveMessage(g, n).B)@Fs]_{g.p'.s',pwdlist'} \\ \Gamma \cup \langle P \cup \{0.p.0, 0.p'.0\}, Sr \cup \{(s', cs, p')\}, Sb, Pb\rangle_{g,pwd} \end{array}}{\begin{array}{c} \Pi, [(suspend().A[out/x])@Es]_{g.p.s,pwdlist}, [(B[g.p.s/x, in/inp])@Fs]_{g.p'.s',pwdlist'} \\ \Gamma \cup \langle P \cup \{0.p.0, 0.p'.0\}, Sr \cup \{(s', cs, p')\}, Sb, Pb\rangle_{g,pwd} \end{array}}$$

The (*invoke(sync)*) rule gives the semantics of the invocation primitive in the synchronous case. The first line expresses the fact that the operation n is part of the service contract cs. The second line models an entity $g.p.s$ which is about to invoke an operation n (with parameters, in) on another entity, $g.p'.s'$. The latter (in the third line) is about to execute `receiveMessage()` in order to receive an invocation message concerning the operation n in the group g (protected by pwd). The fourth line shows that both entities are members of the group g and $0.p'.0$ has published a service s' (with a service contract cs) in g.

In the first line of the conclusion, one can see that both entities have executed their primitives. The code in $g.p.s$ is now "$suspend().A$" which represents A's execution blocking (until a corresponding `reply()` is called). Every free occurrence of x and inp are replaced by $g.p.s$ and in in $g.p'.s'$'s code, respectively. This models the reception of the invocation parameters by $g.p'.s'$. The last line remains the same.

$$\textit{(invoke(async))}\quad \frac{\begin{array}{c} n \in cs \\ \Pi, [(invoke(g.p'.s', n, in).A)@Es]_{g.p.s,pwdlist}, \\ [(\langle x, inp\rangle = receiveMessage(g, n).B)@Fs]_{g.p'.s',pwdlist'} \\ \Gamma \cup \langle P \cup \{0.p.0, 0.p'.0\}, Sr \cup \{s', cs, p'\}, Sb, Pb\rangle_{g,pwd} \end{array}}{\begin{array}{c} \Pi, [A@Es]_{g.p.s,pwdlist}, [(B[g.p.s/x, in/inp])@Fs]_{g.p'.s',pwdlist'} \\ \Gamma \cup \langle P \cup \{0.p.0, 0.p'.0\}, Sr \cup \{s', cs, p'\}, Sb, Pb\rangle_{g,pwd} \end{array}}$$

The (*invoke(async)*) rule gives the semantics of the invocation primitive in the asynchronous case. The condition part of the rule is the same as in the synchronous case. However the conclusion is slightly different. Since a one-way call to `invoke()` does not block the caller, one can see that $g.p.s$ code is not suspended. In contrast, it continues its execution normally.

$$\textit{(reply)}\quad \frac{\begin{array}{c} \Pi, [(suspend().A[out/x])@Es]_{g.p.s,pwdlist}, [(reply(g.p.s, n, out).B)@Fs]_{g.p'.s',pwdlist'} \\ \Gamma \cup \langle P \cup \{0.p.0, 0.p'.0\}, Sr \cup \{s', cs, p'\}, Sb, Pb\rangle_{g,pwd} \end{array}}{\begin{array}{c} \Pi, [A[out/x]@Es]_{g.p.s,pwdlist}, [B@Fs]_{g.p'.s',pwdlist'} \\ \Gamma \cup \langle P \cup \{0.p.0, 0.p'.0\}, Sr \cup \{s', cs, p'\}, Sb, Pb\rangle_{g,pwd} \end{array}}$$

The above rule models the `reply()` primitive. In the first line, one can see that an entity $g.p.s$ is blocked, waiting for the result of a request-response invocation (see (*invoke(sync)*). Another entity, $g.p'.s'$ is about to execute a `reply()` answering to the previous invocation of n by $g.p.s$. The second line shows that both entities are members of g (protected by pwd) and that $0.p'.0$ has published a service s' (with a service contract cs) in g.

The conclusion shows that $g.p.s$ continues its execution, replacing every free occurrence of x by out (the result provided by `reply()`). The last line remains unchanged.

Event management

This paragraph gives the semantics of the event related primitives. In the following, we detail `subscribe()`, `unsubscribe()`, `event()` and `receiveEvent()`. Note that the primitives are

simplified with regard to the model as it is only allowed to (un)subscribe to a specified event in a specified group[13].

$$\textit{(subscribe)} \quad \frac{\begin{array}{c}\Pi, [(\mathit{subscribe}(g,n).A)@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [A@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr, Sb \cup (g.p.s,n), Pb\rangle_{g,pwd}\end{array}}$$

The (*subscribe*) rule models the subscription of an entity $g.p.s$ to an event n published in a group g. Its first line shows the caller entity is about to execute `subscribe()`. The second one shows the peer containing the entity (or the entity itself if it is a peer) is member of the group g.

The conclusion models the addition of a subscription. One can see that a pair $(g.p.s, n)$ is added to g's Sb set in the second line of the conclusion. In the first line, the primitive is simply removed from the code to be executed.

$$\textit{(unsubscribe)} \quad \frac{\begin{array}{c}\Pi, [(\mathit{unsubscribe}(g,n).A)@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr, Sb \cup (g.p.s,n), Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [A@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}$$

The above rule gives the semantics of `unsubcribe()`, which is the dual of `subscribe()`. The first line shows that the entity is about to call the primitive, willing to unsubscribe from an event n in a group g protected by *pwd*. In the second line, one can see a pair $(g.p.s, n)$ is in g's Sb set, representing the subscription to remove.

As a result, the rule shows that the pair $(g.p.s, n)$ is removed from Sb (in the second line). The first line simply removes the primitive from the code.

$$\textit{(event)} \quad \frac{\begin{array}{c}\Pi, [(\mathit{event}(g,n,\mathit{input}).A)@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [A@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0\}, Sr, Sb, Pb\rangle_{g,pwd} \\ \forall\{(g.p'.s', n)\} \in Sb : \exists\{(g.p.s, g.p'.s', n, \mathit{input})\} \in Pb\end{array}}$$

The (*event*) rule models the release of an event n (with some payload, *in*) in a group g. The first line models the primitive call with its parameters, while the second one shows that the entity is member of the group g, protected by *pwd*.

The first line of the conclusion simply removes the primitive call from the code. The second one remains unchanged. The third line gives conditions on $P \in \Gamma$. This line models the fact that a 4-tuple $(g.p.s, g.p'.s', n, \mathit{input})$ is added into Pb for each entity being subscribed to n.

$$\textit{(receiveEvent)} \quad \frac{\begin{array}{c}\Pi, [(\langle x, in\rangle = \mathit{receiveEvent}(g,n).A)@Es]_{g.p.s,pwdlist}, [B@Es]_{g.p'.s',pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0, 0.p'.0\}, Sr, Sb \cup \{(g.p.s,n)\}, Pb \cup \{(g.p'.s', g.p.s, n, \mathit{input})\}\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [(A[\{\mathit{input}/in, g'.p'.s'/x\}])@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P \cup \{0.p.0, 0.p'.0\}, Sr, Sb \cup \{(g.p.s,n)\}, Pb\rangle_{g,pwd}\end{array}}$$

The above rule gives the semantics of `receiveEvent()`. One can see in the first line that the entity $g.p.s$ is about to call the primitive and store the results in $\langle x, in\rangle$ where x is the provider of the event and *in* is the payload. This line also models that another entity $(g.p'.s')$ is active. The second line shows that

- two entities $0.p.0$ and $0.p'.0$ are members of g (protected by *pwd*) (i.e. $0.p.0$ and $0.p'.0 \in P$),

[13] In the SMEPP service model, one can subscribe to all events in a group, or an event in all groups or all events in all groups.

- $g.p.s$ is subscribed to n, $Sb \cup \{(g.p.s, n)\}$,
- the event provider $(g.p'.s')$ has published an event n (with a payload $input$).

The two last points imply that a 4-tuple $(g.p'.s', g.p.s, n, input) \in Pb$, since such a tuple is added to Pb for each subscribed entity when an event is published.

The conclusion shows that the result of the primitive are replaced in $g.p.s$'s code, in the first line. The second line shows that the instance of the event intended to be received by $g.p.s$ is removed from g's Pb's set.

Exception composition

This paragraph gives the semantics of the program behaviours when exceptions occur.

$$\textit{(parallel(1))} \quad \frac{\begin{array}{c}\Pi, [(A \parallel raise(e, in).B)@Es]_{g,pwdlist} \\ \Gamma \cup \langle P, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [raise(e, in)@Es]_{g,pwdlist} \\ \Gamma \cup \langle P, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}$$

$$\textit{(parallel(2))} \quad \frac{\begin{array}{c}\Pi, [(raise(e, in).A \parallel B)@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [raise(e, in)@Es]_{g,pwdlist} \\ \Gamma \cup \langle P, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}$$

Those two first rules show what happens when two programs run in parallel (inside the same entity), surrounded by a same `FaultHandler` ("@" notation), and one of them raises an exception "e". The rules state that both programs are stopped by this exception being raised. In the first line of each rule, one can see an entity having two programs running in parallel, surrounded by the same `FaultHandler`. The second line shows simply that the entities are in a group.

The conclusions are the same for each rule. Both A and B got their execution stopped. It only remains the raising of an exception e surrounded by the `FaultHandler` Es.

$$\textit{(append)} \quad \frac{\begin{array}{c}\Pi, [A@Es@Fs]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [A@Es + +Fs]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}$$

This rule (*append*) gives the semantics for nested `FaultHandler`s. The first line models an entity having a program surrounded by a `FaultHandler` Es itself surrounded by another `FaultHandler`, Fs. In the conclusion, one can see that they are merged in a single `FaultHandler` composed by Es and Fs (using ++ operator), keeping the same order.

$$\textit{(throw)} \quad \frac{\begin{array}{c}\Pi, [(throw(e, in).A)@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}{\begin{array}{c}\Pi, [raise(e, in)@Es]_{g.p.s,pwdlist} \\ \Gamma \cup \langle P, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}$$

$$\textit{(peerNotInGroup)} \quad \frac{\begin{array}{c}\Pi, [(leaveGroup(g).A)@Es]_{0.p.0,pwdlist} \\ \Gamma \cup \langle P, Sr, Sb, Pb\rangle_{g,pwd}; (P \cap \{0.p.0\}) = \varnothing\end{array}}{\begin{array}{c}\Pi, [raise(peerNotInGroup, someInput)@Es]_{0.p.0,pwdlist} \\ \Gamma \cup \langle P, Sr, Sb, Pb\rangle_{g,pwd}\end{array}}$$

The two last rules provide the semantics of an exception raising. When an exception is raised, either with calling `throw()` or when a primitive misbehaves, the rest of the program code is replaced by "$raise(e, in)$" which models the triggering of exception handling.

The second rule gives an illustration of exception modelling. The ($peerNotInGroup$) rule models the misbehaving of the `leaveGroup()` primitive when a peer wants to leave a group it does not belong to.

For the readability of the document, we did not include the rules corresponding to all the primitives. Indeed, that would have needed to write a rule for each exception throwable by each primitive. Note that all these rules are simple adaptations of the above rules.

Exception handling

This paragraph gives the semantics of exception handling. The three rules show that the catch list is processed in sequential order. In case the ($caught$) rule is applicable, the program code is replaced by the code associated with the catch. If no corresponding *catch* clause can be found, the program exits.

$$(caught) \quad \frac{\begin{array}{c} E = catch(e, x).C \\ \Pi, [raise(e, in)@E : Es]_{g.p.s,pwdlist} \\ \Gamma \end{array}}{\begin{array}{c} \Pi, [C[in/x]]_{g.p.s,pwdlist} \\ \Gamma \end{array}}$$

This rule shows the case in which the first catch clause of the `FaultHandler` matches the raised exception e. The code C related to the catch clause E is then executed.

$$(uncaught) \quad \frac{\begin{array}{c} E \neq catch(e, x).C \\ \Pi, [raise(e, in)@E : Es]_{g.p.s,pwdlist} \\ \Gamma \end{array}}{\begin{array}{c} \Pi, [raise(e, in)@Es]_{g.p.s,pwdlist} \\ \Gamma \end{array}}$$

The above rule is the dual of the first one. The catch clause does not match the raised exception e, the processing of the `FaultHandler` continues with the rest of the list, Es.

$$(stop) \quad \frac{\begin{array}{c} \Pi, [raise(e, in)@[]]_{g.p.s,pwdlist} \\ \Gamma \end{array}}{\begin{array}{c} \Pi, [exit()]_{g.p.s,pwdlist} \\ \Gamma \end{array}}$$

This last rule shows that if no matching catch clause is found, the program exits. In the first line, one can see the `FaultHandler` is empty (this is modelled by "[]"). The $exit()$ operation represents the termination of the program in $g.p.s$.

2.4 Scope of the thesis

As part of our master thesis, we had the opportunity to work with the team of Professor A. Brogi from the University of Pisa, in the SMEPP project. During our internship in Pisa, our objective was to develop a proof-of-concept of the SMEPP service model. To be more precise, the goal was to create a (prototype) software which takes as input a piece of code written in SMoL and outputs an executable program. The implementation was twofold. An API of the SMEPP primitives would provide a basic SMEPP middleware, while a SMoL translator using this API would produce executable code.

In the four months devoted to our stay in Pisa, it was not reasonable to think of implementing the SMEPP middleware. Hence, in a mutual agreement with the team in Pisa, we decided that the prototype implementation would have the following limitations.

- SMEPP intends to provide wrappers to allow any kind of program to be part of a SMEPP application. We decided that our implementation would only consider peers and services defined in SMoL.
- SMoL being a specification language and our goal being to produce executable code, we decided that some additional coding would be needed by the programmer after the translation, to make the program actually executable. In particular, SMoL envisions to use XPath to describe data. We decided to simplify the data representation using traditional XML flags and strings.
- The SMEPP service model borrows concepts from the web service technologies. In particular, service contract are matched[14] by using the Web Service Description Language (WSDL [109]) and ontology information defined with the Ontology Web Language (OWL [108]). The problem of service contracts matching is a very important problem in itself and we decided it was out of the scope of our work. Our implementation would only consider syntactical comparison of contracts.
- We decided our implementation would only implement the default level of security of SMEPP (*level 1*). This is justified by the fact that this level is the de facto standard and that the higher security level was still not well defined. Note that the no security level (*level 0*) is implicitly covered in *level 1* since it suffices to use "empty" keys.
- Finally, our goal is to emphasise on coordination issues and not on platform-specific ones. For instance, we do not address the problems induced by motes and other sensors (such as little battery autonomy, lack of computing power etc.).

The objectives of our work being defined, the next step is to select a suitable coordination language on top of which the SMEPP proof-of-concept can be built. In order to direct the research, we elicited the requirements we expect from this language. The next section details them.

2.5 Middleware requirements

This section intends to elicit the requirements that the target coordination language has to meet. To build this list we take into account the key features of SMEPP and the goal of our master thesis. The rest of the section details each requirement.

R1 Peer-to-peer orientation. As the SMEPP project is by essence peer-to-peer oriented, we have to find a decentralised middleware. That is, the system cannot rely on a pre-existing infrastructure (i.e. no centralised server). Only the peers form the network. Furthermore, the project aims to be used in embedded environment, where devices can join and leave the network whenever they want (depending on the peers' will and the availability of the connection). Thus, we have to find a middleware which handles transient connection of peers. This means the system has to feature mechanisms to take into account the fact that a connection between two peers can be lost at any moment.

R2 Security. SMEPP defines a configurable model of security which, by default, uses symmetric keys to manage access rights: one preshared symmetric key to access a SMEPP application (i.e. to become a SMEPP peer) and one preshared symmetric key for each group. In addition to access control, SMEPP defines group and service visibility restrictions. A peer can only discover groups and services of which it has the corresponding key.

In order to implement this model of security, the coordination middleware has to provide some mechanisms to protect the data transferred between two peers. It also has to restrict the access to the communication to authorised peers. Since we intend to implement the default level of security, simple mechanisms using symmetric keys could be enough.

[14] See `getServices()` in Subsection 2.3.1.

R3 Available implementation. The purpose of our work is to implement the SMEPP service model by using a coordination language, not to make a new implementation of an existing language. Having an executable language gives us a twofold opportunity. On the one hand, it allows to prove the service model is usable in a real environment. On the other hand it shows the suitability of coordination languages to implement a complex middleware.

R4 Java-integrability. The reference implementation of the SMEPP project will be Java-based. This means that the complete implementation of SMEPP will be developed with the Java programming language. This implementation intends to be used on computers (smaller devices such as mobile phone or mote would require a particular implementation of the SMEPP middleware). We decided to develop our implementation using Java, in order to give useful feedback from our proof-of-concept, in particular with regards to the local management of peers and services. Thus, we needed a middleware providing Java integrability (i.e. a Java-based API).

The four previous items are, of course, a minimal set of requirements our target has to meet. A language providing more relevant functionalities, such as asymmetric keys based security mechanisms or implementation running on embedded operating systems, would provide additional ways of extending our implementation.

Chapter 3

State of the art

This chapter intends to achieve two goals. The first is to make the reader comfortable with what coordination languages are and to highlight that the underlying abstract model of tuple space based languages suits well the implementation of complex distributed systems. This first objective is covered by Section 3.1 which describes the families of coordination languages and which explains why one of them is well adapted to open systems. The second goal is to survey tuple space based coordination languages according to their features and the coordination issues they intend to solve, in order to choose one of them to implement our SMEPP proof-of-concept. Section 3.2 highlights the main issues having to be tackled by them and explains how these issues intervene in the context of SMEPP. Section 3.3 presents a general classification for these languages. Buildt upon the two previous ones, Section 3.4 presents a framework helping us to choose a relevant language according to the SMEPP middleware requirements presented in Section 2.5. Finally, Section 3.5 describes the implementation of the chosen coordination language.

3.1 Coordination models, languages and systems

This section, essentially based on [85], aims firstly at making clear what is meant by coordination languages. In order to do this, we must first introduce the concept of *coordination model*, which is, according to [82], the basis of two notions: *coordination languages* and *coordination systems*. A *coordination language* represents a linguistic reification of a coordination model, whereas a *coordination system* provides a programming environment, an architectural framework, or an infrastructure for the model implementation.

The notion of *coordination model* introduced in [82] can be defined according to two different points of view. The first defines a coordination model as a formal framework for expressing the interaction among components in a multi-components system [32], whereas the second, closer to [51], defines a coordination model as a conceptual framework for shaping the space of component interaction. Although these two definitions can seem very similar, they are representative of two different points of view, respectively the one of computer science and the one of computer engineering. The first definition represents a coordination model as a formal framework providing symbols and rules to model the coordinated systems and all the interactions occurring in them. The main purpose of this approach is to provide computer scientists with theoretical tools to model, analyse and validate properties of the interaction space.

The second definition represents a coordination model as an abstraction helping developers to effectively manage the space of inter-component interactions. This definition makes sense because, when considering the engineering of Internet-based multi-agents systems, interaction can be recognised as an independent dimension. Gelernter and Carriero formulated in [51] a simple equation claiming the separation between the specification of the components and the

specification of their interactions. This equation is the source of the term *Coordination* in computer science:

$$Programming = Computation + Coordination$$

meaning the process of separating computation from communication concerns. When taking this second coordination model definition into account, coordination models focus on effectively easing engineers' task to manage complex interactions in multi-components systems, rather than to focus on formal properties of the model.

Various coordination models have been, at first, exclusively designed by taking into account the first definition ([31, 66, 81, 115]), or the second (the most famous being Linda [50], but also [35, 48, 83, 113]). However, the absence of formal semantics has led to inconsistent implementations of, at first sight intuitive, primitives functions dealing with the interaction space of the components [21]. Consequently, we would like the reader to be aware that a fundamental issue of the research on coordination is to take both definitions into account. Indeed, because coordination models and languages are more and more used in the engineering and modelling of complex systems of tomorrow, especially in the context of agents societies interacting with each other through open environments such as the Internet. Such applications (multi-agent- and/or Internet-based) need to cooperate, coordinate and share their information with other applications, either with or without user intervention. They interact in an environment where data and resources are distributed. Therefore, it is really important to develop suitable coordination technologies [86].

Currently, coordination solutions are proposed at different levels. One proposal, referenced by [86], is that the basic services including coordination of activities and information should be provided directly by the Web infrastructure. This results in the development of coordination architectures for building collaborative applications. A more classical approach is to design a coordination language and its associated coordination model to offer programming notations, providing solutions to the problem of specifying and managing the interactions among computing agents. Actually, they offer mechanisms to compose, configure and control architectures made of independent or distributed active components [20]. The fact is that coordination models are often used as an ontology for agent-based software design, taking their purpose of managing active components architecture into account. Currently, the designers of such models are often interested in defining the concept of an agent, which can be mobile, autonomous, "intelligent", etc.

This point of view, developed in [20] seems to be true when looking at different projects dealing with the coordination of entities in open environments (e.g. POPEYE [89], WORKPAD [90] and SMEPP [98]). For instance, SMEPP peers can be considered as mobile agents searching groups or services to respectively join or invoke them. Thus, as SMEPP matches the definition of a coordination model, it is relevant to survey existing coordination languages and systems in order to implement it.

Existing coordination systems in general have a common objective: to facilitate the interaction of applications, programs or agents executing on distributed heterogeneous environments. Because of this objective, they are often referred to as *middlewares*, situated between the application and the network layers, helping the abstraction of the interactions by hiding details related to the physical properties of the environment. These middlewares differ mainly in the way they model the communication and in the communication related services they offer.

In [86] and [94], two classifications of coordination systems are proposed. Although they introduce different levels of classification, we can identify two similar categories in both approaches: *basic coordination infrastructures* and *coordination frameworks*. The former only introduces the elementary enabling technologies for building coordination systems. These systems focus essentially on communication and can be seen as the lowest level of coordination. The latter, focuses on the coordination activities by providing mechanisms to model them and

by considering them as its core elements. This approach deals with issues particularly relevant to Internet-based applications. For instance, providing ways of searching, reading and deleting information in distributed data collections. The "coordination level" of this approach is considered higher than the one of *basic coordination infrastructures* as it offers richer coordination mechanisms.

In order to determine which of these approaches is best-suited to the SMEPP middleware, we present the two most widely used [94] interaction mechanisms: message passing and remote procedure call (RPC), that we classify under *basic coordination mechanisms*. Then, we present the tuple space based coordination systems that we classified under *coordination frameworks*.

3.1.1 Coordination systems based on the message passing and RPC paradigm

The services of message passing based coordination systems focus on the emission and reception of messages. They help to pack information into messages understandable by heterogeneous applications, by proving a standard Application Program Interface (API). A standard API enables the same source code to be compiled on many different platforms. Another common service of message passing systems is to help managing each of the network members' messages queues. Well supported and mature examples of such systems [94] are PVM [103] and MPI [47]. Although a good programmer using message passing systems can produce very efficient distributed programs, the task is not really easy. Rowstron (in [94]) compares that difficulty to the one of producing better machine code than a compiler. These systems are very low-level and thus, provide only basic coordination mechanisms. Using message passing in order to implement SMEPP could be an efficient solution, but does not abstract enough coordination mechanisms to ease its development.

Remote Procedure Call (RPC) [9, 76] also focuses on easing communication between heterogeneous applications. RPC-based middlewares communication abstraction is to consider a method invocation on a distant computer as the one of a locally invoked method. The middlewares will provide services to pack methods parameters and to send them through the network. In addition, they also provide services to retrieve the result of an invocation. RPC has become very popular [94] and is used in DCOM [64], CORBA [54] and Java RMI [65]. Common RPC applications are designed in a client-server form: a program provides services that a client can invoke by getting in contact with the server. However, it is possible to design applications acting as both client and server, so SMEPP applications could be developed using RPC. Still, we consider RPC-based middlewares being at the same level than message passing systems. They offer slightly richer coordination mechanisms helping for instance to retrieve results of methods invocations, saving some efforts regarding the emission of messages, but using RPC-based systems to develop the SMEPP middleware is still rather complex.

For instance, modelling group communications and catching exceptions thrown by peers disconnections is much more difficult using RPC than using a coordination framework. Here, a group member, in order to send a message to the group, would have to call a procedure on a host (himself or a distant host) being responsible of the message publication. The message publication process would have to send the message to each group member. In most RPC implementations, the communication is *synchronous*, meaning that the sender and the receiver are bound during a period required to communicate. Exceptions could occur during this process: one of the invoked hosts could disconnect and threads could stay interrupted waiting for an answer because of the communication synchrony. Therefore, one can see that components communicating with each other are tightly coupled in space and time. However, the high dynamics in ad hoc networks raises the necessity of employing a communication paradigm that decouples application components in space and time [57].

The discovery of peers, groups and services would require a way to model an entity, which would have to be contacted at any time, responsible of registering the available groups, peers and services. CORBA NamingService [53] could for instance be used to achieve this goal, but the fact that SMEPP requires a decentralised approach implies that a peer cannot know

in advance all the possible peers and the objects they could register in the NamingService. Furthermore, the SMEPP decentralisation requirement comes into conflict with the centralised implementation of the CORBA NamingService.

In the light of the difficulties to implement the SMEPP middleware with *basic coordination infrastructures*, we now present the tuple space paradigm, that we classify under *coordination frameworks*.

3.1.2 Coordination systems based on the tuple space paradigm

Currently, different approaches are proposed, providing richer coordination models than message passing and RPC. However, the systems implementing them are often based on one of these two paradigms. In general, coordination frameworks are not standalone, general purpose, programming languages. They are often designed as language extension and focus only on coordination issues [20]. Examples of such coordination models include event based systems, distributed shared memory or objects, tuples or data spaces systems as in Linda [50], various forms of "multiset" rewriting as in Gamma [5] and models with explicit support for coordinators as in Manifold [3] and Reo [4]. The fact is, according to [94], that none of the middlewares based on these models is likely to become *the* reference middleware because each of them suits *different applications*. For instance, it is quite difficult to conceive video streaming through shared spaces, but control information could be retrieved from a tuple space in order to stream videos.

While a great number of models have been designed, a few have been as successful as Linda's data space model [50]. Implementations extending the basic ideas of this model have become ubiquitous and are widely used in the context of web-based applications [94]. The popularity of this model, which has been the first considered as a coordination language, is such that the notion of coordination language is often confused with it.

In order to explain the suitability of the data space model to implement distributed open systems such as SMEPP, we present here the key concepts of Linda and the main advantages of the tuple space paradigm.

The key concepts of Linda are a shared data space, called *tuple space*, where *tuples* can be put or retrieved from, and a small set of *coordination primitives* providing means to manipulate the tuple space. The primitives are orthogonal to any particular language, they are part of the coordination language and can be added to any other computation language [93].

The main primitives are `out`, `in` and `rd`. They respectively allow to put a tuple on the tuple space, to retrieve (and remove) a tuple matching its parameter from the tuple space and to read the content of a tuple matching its parameter. Matching is determined by a *matching rule*: a *template* is provided as parameter of the `in` and `rd` primitives, and the tuple matching this template is returned. In the case of multiple tuples matching the template, Linda does not specify which tuple is retrieved. Here is an example of `out` and `in` primitives to understand better the matching process: `out(<"string",10,a>)` puts a tuple containing three fields (a string, an integer, and the content of the program variable `a`) to the tuple space. Now, if `in(<"string",10,?b>)` is called, the tuple matching the template `<"string",10,?b>` will be a tuple having its first field containing the string "string", its second field containing the integer 10 and its last field containing a value of the same type as the program variable `b`. The notation `?b` indicates that the value has to be bound to the variable b after retrieval.

Both `in` and `rd` operations are blocking, they suspend the execution of the program as long as a matching tuple cannot be found. Non-blocking versions of these primitives are `inp` and `rdp` returning true when a matching tuple is found and false otherwise.

The last Linda primitive is `eval`. Its role is to emit an *active tuple* in the tuple space, a tuple where one or more fields do not have a definite value, having to be computed by function

calls. When an *active tuple* is emitted, a new process is started for each of the fields having to be computed. If all these processes finish their computation, this tuple is replaced by a "normal", or passive tuple, where the to-be-computed fields are replaced by the results of the processes. This primitive enables dynamic creation of processes in Linda systems [93].

As we can see, the key concepts of Linda are pretty simple and there is only a small set of primitive functions. However, it has been demonstrated that Linda is expressive enough to build parallel applications [28], to express major styles of coordination for parallel applications [27], to design distributed computing platforms [111, 113] and to program agent-based systems [34, 37]. This demonstrates the expressiveness and the relevance of tuple-space systems, which are much more than simple and elegant.

D. Rossi, summarises in [93] the advantages of Linda-based systems for programming open application:

- **Uncoupling**. Using a tuple space as a coordination medium uncouples the coordinated components both in space and time. The agent performing an `out` in the tuple space does not have to care about the presence of the agent that will retrieve it. This agent can even terminate before the tuple will be retrieved. This feature added to the fact that agents do not have to be in the same location to interact with the tuple space eases the task of managing interactions in open environments.

  For instance, in SMEPP, this feature could be of great use to handle arbitrary peer disconnection.

- **Associative addressing**. The way tuples are retrieved using templates specifies *what kind* of information is requested, rather than *which* tuple. This enables more abstract ways of retrieving information than just retrieving a specific message.

  This feature could be very useful in the context of SMEPP to retrieve lists of services or groups matching user's criteria.

- **Asynchrony and concurrency**. The tuple space abstraction can be used to model parallel applications and can ease the way of dealing with processes concurrency [28, 27].

- **Separation of concerns**. Coordination languages as Linda are not influenced by characteristics of the host programming language.

  In the context of SMEPP, this is an important feature as it targets the interaction of peers running in heterogeneous systems and invoking services built with different programming languages.

The tuple space abstraction provides an easy way of mapping specific SMEPP concerns. For instance, one could model group communication by using different tuple spaces or by using partitions of the same tuple space. And, one could model an event emission by sending a tuple corresponding to it in a tuple space. Thus, the tuple space paradigm seems particularly well suited to SMEPP applications and appears as an interesting way to ease SMEPP implementation compared to RPC or message passing middlewares. However, there are emerging coordination issues to be solved when tuple space based coordination systems are used.

3.2 Emerging issues of coordination

This section aims at presenting the emerging issues of coordination in tuple space systems and to show their implications in the SMEPP middleware implementation. Presenting these coordination issues also highlights the required features of coordination systems that could be used in order to develop the SMEPP middleware.

3.2.1 Run-time systems implementations concerns

Every tuple-based implementation requires a run-time system, often called *tuple-space manager*, *TS Manager*, *TSM*, etc. We can compare it to the *kernel* of the system [94]. The designers of kernels take design decisions influencing the purpose of the systems. The most relevant feature to study is whether implementations are *open* or *closed*, as programming techniques to implement these two kinds of systems are very different. A *closed implementation* is an implementation which needs all the information about entities that will communicate through the tuple space at the kernel startup. In such systems, it is therefore impossible to join or leave the network at will. This kind of system does not suit at all the SMEPP requirements. In contrast, an *open implementation* does not require the kernel to have this kind of information. The kernel in such systems is distributed as a set of *kernel processes* among the entities of the system. In these systems, entities can join and leave the network at will. Closed implementations suit better parallel computing while open implementations are well adapted to Wide Area Network (WAN) based applications and ad hoc networks. Usually, open implementations do not emphasis on efficiency (as closed implementations do) but on supporting security, reliability, heterogeneity and availability [94]. This leads to different techniques to implement both kinds of systems.

All open implementation kernels require a number of design choices to be made. We want to focus on the following choices:

- *Tuple distribution*: how are the tuples going to be distributed across kernel processes?
- *Tuple format*: how are the tuples represented?
- *Tuple storage*: how are the tuples stored within a single kernel process?

Tuple format and storage mainly influence the performance of the kernel. There is usually a tradeoff between the complexity of the data structure chosen to store tuples within a process and the cost of comparing tuples and templates [94]. For instance, the number of tuples to be checked in order to find a tuple matching a template can vary according to tuple storage choice. The bigger the number of checks to perform is, the higher the cost of the communication is. One can reduce the cost of checking tuples against templates by choosing a simple tuple format. It is also important to define whether the language needs a language-independent way of encoding the tuples in case the language aims at supporting various host languages.

We want now to focus on explaining the different tuple distribution techniques. Four techniques are presented in [94]:

- **Centralised**. A unique kernel is called by every processes to perform tuple-space operations and all the tuples are stored in it. The advantage of this approach is that the kernel is kept simple and that it is easy to know the state of the tuple space at any time. The main problem of this approach is that the single kernel becomes a bottleneck and a point of failure when a high number of processes perform concurrent tuple space operations. This approach is not well suited to large-scale environments but is more adapted to a small set of processes having to communicate via the tuple space.
- **Uniform Distribution**. There is more than one kernel and the tuples are distributed evenly over them. They are often designed in such a way that every application communicating in the tuple space owns two kernel processes: an *in-set* and an *out-set*. When a tuple is sent to a tuple-space, it is broadcasted to all the kernel processes' *out-set*. When a tuple is required from a tuple space, the request is broadcasted to all the kernel processes' *in-set*. In the case of an *in* operation, kernel processes have to synchronise their contents in order to prevent the same tuple to be retrieved twice. This approach suits more the concept of distributed and decentralised applications such as SMEPP but can be enhanced in terms of performance using the Intermediate Uniform Distribution approach.

- **Intermediate Uniform Distribution**. This is a particular form of Uniform Distribution where the cardinality of the *in-* and *out-set* is equal to the square root of the number of applications (nodes) communicating in the tuple space. For instance, if 16 nodes are used, there will be 4 *in-sets* and 4 *out-sets*, meaning that *in-sets* and *out-sets* will be shared. The sharing is done in such a way that 4 nodes share the same *out-set* but these same nodes do not share the same *in-set*. This form of distribution has been proved optimal in terms of the number of nodes involved in an *in* primitive and an *out* primitive [1]. This approach is used in the Linda machine [1, 62], providing hardware means to ensure the consistency of the tuple spaces after several *in* primitives. When used without specific hardware, as in [46, 104], it is shown [46] that the cost of synchronisation is too high. Indeed, kernel processes of the *in-set* concerned by an *in* operation must first agree on which of them will answer the query, then, the elected one must inform the kernel processes of its *out-set* that the tuple is being removed, then, these processes will have to synchronise their contents. This can easily lead to inconsistencies in the tuple spaces because of the high number of messages being sent between different processes.
- **Distributed Hashing**. This technique is also used in the context of distributed kernels. The kernel process in charge of a tuple is selected upon properties of the tuple itself. A hashing function is applied to the tuple or to a template, to determine which kernel process will be in charge of it. This saves time to search in which kernel process resides the tuple corresponding to a template because the hashing function applied to the template returns the kernel process in which the corresponding tuple is stored [10]. A good hashing function has to possess two properties: to provide a unique mapping between every tuple and templates matching it to a single kernel process, and to provide a good distribution of the tuples. However, such an approach has only been efficiently achieved on closed implementation systems [94]. For open implementations, efficient algorithms have not been found yet because of the limited amount of information provided within a template and the lack of compile-time analysis of all tuples and the templates used within a system. Still, this approach can be used in open implementations as follows: hashing functions will match a tuple to a unique kernel processes, but templates will be matched to several. As a consequence, the request must be either broadcasted to every kernel processes or sent to a specific one. In the first case, the user application will have to take measures if multiple tuples are returned, while in the second case, the request will have to be forwarded again if a matching tuple is not found in the selected kernel process. Thus, in open implementations, the communication overhead of this approach makes it not as efficient as in closed implementations.

In conclusion, any kind of uniform distribution would be suited to SMEPP as we can consider they give similar results on hardwares not particularly designed to enhance the distribution mechanisms.

3.2.2 Coordination and mobility

The notion of mobility requires that there exist entities performing moves, a space where movements are observable and rules governing the motion. An important aspect regarding mobility is that entities should have ways of observing changes occurring in the space to better perceive their environment and take decisions according to it. This ability is often referred to as context management, i.e. the wordlview of the individual units [91]. In this section, we want to discuss the issues introduced by the above mentioned concepts of mobility when considering tuple spaces as the space in which entities interact. The topics of the issues we want to present are related to *the space of interactions* itself and to *context management*.

As we explained in Subsection 3.1.2, uncoupling is one of the advantages of tuple space based coordination systems. They allow an agent to send a tuple in the tuple space without having to care about the presence of an agent who will potentially retrieve it. Moreover, this kind of systems allows the tuple to be retrieved even after the disconnection of the agent who has sent the tuple. The question is whether this statement holds in the context of mobile ad

hoc networks? The fact that a tuple can be retrieved after an agent's disconnection requires the presence of an always present and shared entity responsible of storing and managing the tuples as agents do not know each other and as they enter and leave the network at will. This does not embody the concept of mobile ad hoc networks and decentralised peer-to-peer networks that cannot rely on a central entity. It is therefore needed to rethink the space of coordination itself in order to be useful in such kinds of applications. A realistic situation requiring this kind of communication could be cars on a highway exchanging information among themselves when being physically close enough. Interesting models have been proposed to deal with mobility, in particular, Lime [75], a middleware designed to provide transient and transitive sharing of tuple spaces. In this model, "mobile agents reside on mobile hosts that can form ad hoc networks when in proximity of each other. When this happens, the agents appear to share a common data environment (tuple space) and have the opportunity to jump from one to another [91]". We shall not detail here the concepts of Lime, as this will be done in Section 3.3 along with other tuple space based systems.

Another issue when considering this kind of mobile applications and tuple space based coordination systems is *context management*. How can mobile agents be aware of the presence, arrival and departure of new agents in the network? Such a notification feature could be implemented using the tuple space model by adding a tuple containing presence information in a shared tuple space. This solution induces again a central entity which is not appropriate to mobile applications. Moreover, the management of the presence information can be difficult in the case of sudden disconnections of an agent. In this case, the agent cannot communicate its new status to the central entity. Therefore it should be responsible of checking at regular intervals of time the presence of agents.

The SMEPP middleware has to deal with the above mentioned issues induced by mobility. Indeed, peers must discover groups and services in a unpredictable environment. They are also responsible for the consistency of the data sent which is an additional complication in the case of sudden disconnections. It is therefore important to chose a tuple space based coordination language providing means easing context management in a decentralised environment.

3.2.3 Coordination and security

So far, we did not introduce the notion of security in tuple spaces. In the current situation, there is no guarantee that a tuple sent to a tuple space by an agent A to an agent B would be actually retrieved by B. This issue, together with other security issues having to be tackled by tuple space systems, are presented in this section by introducing an architectural solution proposed by [16]. While explaining the concepts of this architecture, we present its suitability in the SMEPP security model.

We start by presenting the above mentioned general security architecture for tuple space systems. A secure system must possess a *trusted computed base* (*TCB*) [29]. A *TCB* is the set of code and data needed to secure system operations. It can be used to store access rights, encryption keys and to house authentication and authorisation procedures. The primary component of a *TCB* is the *reference monitor* [29]. Its role is to intercept each access by an agent to a tuple space to verify that this access is allowed by the *security policy*, being the set of rules specifying how sensitive information has to be accessed in a computer system [29]. Figure 3.1 illustrates a reference monitor for the shared space model. Its role is to intercept every tuple space operation and to consult the security policy in order to allow or refuse it.

The first comment we can make regarding the suitability of this architecture to the SMEPP security model is that the reference monitor cannot be a centralised unit intercepting every tuple space operation. The reference monitor should be local to every peer in order to keep the implementation decentralised. As a consequence, the security policy will not be managed by a trusted central entity. It follows that the security policy must rely on a pre-defined and static basis.

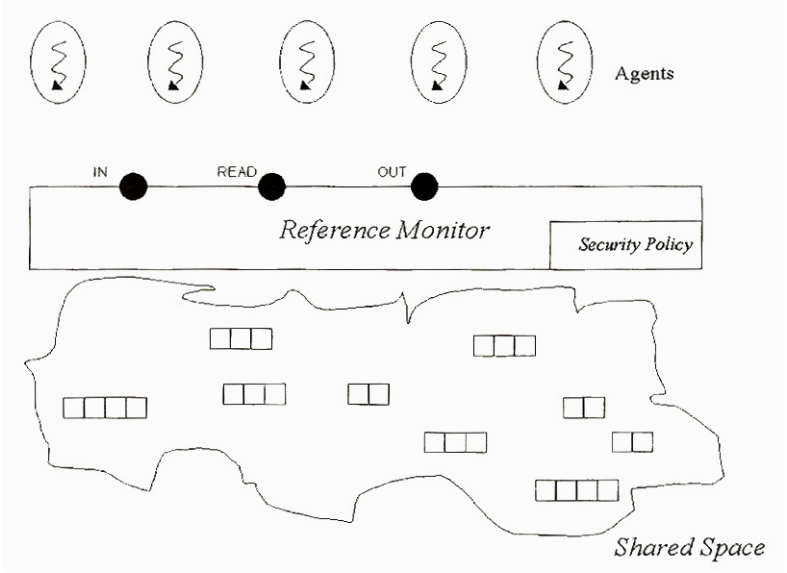


Figure 3.1: The shared space model with a reference monitor.

A *TCB* must also provide mechanisms for identifying agents (*authentication*) and for verifying their privileges in the shared space (*authorisation*) by defining a flexible security policy. Flexibility in the context of security policies means that a wide range of policies can be supported, from a policy allowing or refusing full access to a tuple space to a policy controlling which tuples can be accessed and what operations are allowed on them. Flexible security policies could be used in the context of SMEPP to deal with different security levels giving more or less privileges to specific peers, for instance, granting access to a limited number of groups.

Now that we have introduced the main components required by a *TCB*, we want to present protocols proposed by [16] to manage authentication and authorisation in tuple space based systems.

Authentication is a pre-condition for authorising a request. It is needed to determine the access rights that can be granted to an agent [29]. Authentication is more general in open systems, as an agent can personify one or more individuals. To be authenticated, agents are not forced to show their real identities but are assumed to show a set of data called *credentials* sufficient to recognise them. This idea is the starting point of what Bryce, in [16], calls the *reference* authentication protocol, the *protocol authentication process*, or *PAP*, shown in Figure 3.2. It is called *reference* protocol because of its generality. Basically, a user wishing to be authenticated shows its credentials to the *PAP*. If the authentication is successful, the *PAP* returns an authentication token to the agent. Then, when the user wants to communicate using the tuple space, the authentication token is joined to every of his request. In this way, the reference monitor checks which permissions are associated with this authentication token and decides whether the request is allowed or refused. Either symmetric or asymmetric cryptography can be used to implement authentication protocols in general.

C. Bryce ([16]) proposes an implementation of the *PAP* protocol over tuple spaces that can be summarised as follows:

- It is assumed that each entity and the *PAP* own a pair of keys used for asymmetric encryption. One of the keys is public and the other is kept secret. A data encrypted with a public key can only be decrypted using the corresponding secret key and a data signed with a private key can only be verified by using the corresponding public key. The term signed is used when encrypting with a private key because only the owner of the private key can compute this encryption. Thus, it is considered as the owner's signature.

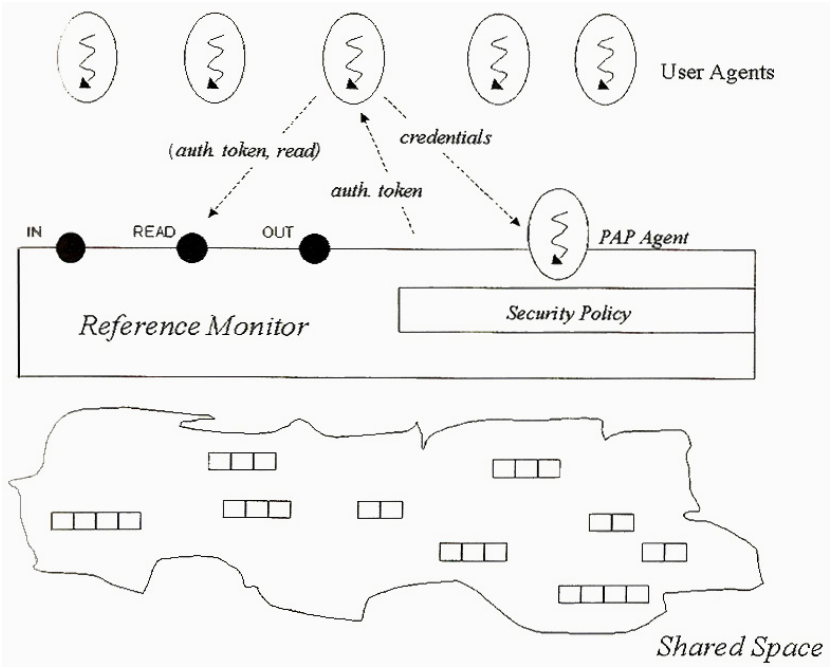


Figure 3.2: The protocol authentication process.

- The communication between the *PAP* and the entities is done using the same tuple space for every entity.
- A client wishing to be authenticated emits a tuple encrypted with the public key of the *PAP* so that only the *PAP* can read it. This tuple contains the credentials of the client signed with its own private key to ensure it has been emitted by him. Note that even if this tuple is retrieved by someone else than the *PAP*, its real content will not be revealed.
- In case of successful authentication, once the *PAP* retrieves the previously sent tuple, he answers a tuple encrypted with the client public key. Thus, only the client can know its real content. This tuple contains the authentication token signed with the private key of the *PAP* so that the client can be sure the token was created by the *PAP*.

Although this implementation suits well systems where a central entity can act as the *PAP*, it is not well adapted to decentralised systems because the PAP must know at startup what kind of authentication token it must grant to every possible user. This is actually impossible in an environment where all users cannot be known in advance. What could be done to keep this kind of implementation in decentralised systems is to gather users in categories to which is associated a specific authentication token to ease authentication management. Moreover, the notion of a centralised PAP does not meet decentralised systems ideas and, consequently, does not fit in the context of SMEPP. As a result, one of our tasks in this thesis is precisely to implement authentication by adapting the reference protocol in a decentralised setting.

As explained before, the authorisation process grants a token determining the privilege of the authenticated entity. The process concerned with defining, enforcing and protecting access rights privileges is called *authorisation*. The main problem, regarding tuple space systems, to store and protect access rights, is that tuple spaces can be accessed by any agent, leaving access rights readable and alterable. One issue to consider, helping to solve this problem, is then the *granularity* of access rights: whether rights are granted to agents for tuples attributes, for tuples, or for spaces [16]. For instance, one could consider a *PAP* tuple space, where agents could only perform read operations to avoid access rights to be deleted. The problem of this solution in a decentralised peer-to-peer environment is that it requires a special entity being

able to perform out operations on that *PAP* tuple space. This concept does not embody the decentralised nature of the environment. Thus, another task in the implementation of the SMEPP middleware will be to determine how to deal with authorisation in a decentralised way. In the SMEPP environment, there is the need to bind a peer to its access rights, being the groups it can join and the services it is allowed to discover. This binding is referred as a *Capability* in [16]. The issue is to define, as we said earlier, how these capabilities can be safely stored in a tuple space, but also how to manage their creation: can capabilities be dynamically modified or do they have to be static? The issue of allowing dynamic modifications of capabilities in a decentralised peer-to-peer environment where peers are untrusted seems to be unsolvable if we do not accept that some trusted peers or stationary security entities exist. Therefore, adopting a static way of managing capabilities seems to be the most attractive solution in this kind of environment, as SMEPP security guidelines enforce the absence of trusted peers or stationary security entities.

3.3 A taxonomy for tuple space based systems

Sections 3.1 and 3.2 have highlighted the relevance of tuple space systems to implement open systems and have explained coordination issues to be tackled by them as well as some limitations of the Linda model. To overcome these limitations, many projects have extended the Linda model under different aspects and designed tuple space coordination systems implementing the modified models. This section presents a number of tuple space systems by using a taxonomy, proposed in [93], in order to determine the systems we could use to implement the SMEPP middleware proof-of-concept. Three categories are proposed:

- **Systems extending the primitives**. These systems extend the set of Linda primitives by adding new ones or modifying existing ones. Such systems are often specifically designed for a certain context.

- **Systems adding programmability**. These systems give the capability of modifying the behaviour of the tuple space by programming it, without changing the set of primitives.

- **Systems modifying the model**. These systems enhance Linda by modifying its model. Again, these systems are often designed for specific contexts and purposes.

We present the languages falling into these three categories using a set of criteria, proposed in [93], wide enough to include most aspects of tuple space models and small enough to include only the relevant ones. This set is based on the authors' experience in designing, developing and using tuple space systems. This set can be summarised as follows:

- **Extensibility** considers the following properties: can new coordination primitives be added? Can the behaviour of the primitives be changed? Can the matching mechanism be customised?

- **Data space structure** considers properties of the data space: can one or more tuple space be defined? how are the tuple spaces created and distributed? In the case of multiple spaces, are they considered as a flat collection or hierarchically organised? What kind of data are exchanged via the space?

- **Platform-related issues** consider the influence of the tuple space system host language and its architecture on the coordination language. They also consider the portability of the system and what kind of run-time system is used.

- **Technological Additions** consider transaction support, mobility support, security support and development/analysis tools support.

3.3.1 Systems extending the primitives

Jada

Jada [36, 92], developed by D. Rossi and P. Ciancarini, is a coordination language for Java designed to coordinate parallel and distributed components. The Jada model extends the Linda model by adding new primitives and by allowing the creation of more than one `ObjectSpace`[1], a specialised object container. Local concurrent programming can be achieved by using the `jada` package while distributed programming API resides in the `jada.net` package [92]. The additional jada primitives are: `readAll, inAll, getAll and getAny`. Respectively, the first two read and retrieve *all* the objects matching *one* template, while `getAll` returns all the objects matching *a set* of templates and `getAny` returns *any* object matching *a set* of templates. Every primitive can be used with a timeout, being the time within which a primitive must be executed. Additionally, `out` primitives can specify a time-to-live in order to express the time that the data sent must remain in the `ObjectSpace`. All the *in-like* and *read-like* primitives are non-blocking. They output an instance of the `Result` class, serving the purpose of checking the actual result of the primitives: either a failure (in case of a timeout or by process abortion), or the data result. The retrieval methods on the `Result` class are blocking, until the result is actually available. The Jada matching policy is customisable thanks to the `JadaObject` interface. The `matches` method must be implemented to define how a template matches a tuple.

Distributed programming in Jada is inspired by the client/server paradigm. `ObjectServer`s store `ObjectSpace`s and listen on a port waiting for client request. `ObjectClient`s are created given the hostname and the port of `ObjectSpace` they want to communicate with. This approach cannot be used in a decentralised ad hoc network environment where agents cannot speculate on the presence of other agents and cannot rely on a central entity managing the communication. Security in Jada can be done at two levels: at the tuple space level by defining access control policies and at the communication level by supporting encryption of the content being transferred. Jada has been used for several research projects [93] to implement quite different systems, from parallel computing to Internet-based card games, from distributed collaborative applications to mobile agents systems. Code mobility in Jada is possible as Jada implements the `eval` primitive, `evaluable` objects are moved to a remote JVM in order to be executed as new threads on it. Jada has also been used to provide coordination for applications based on PageSpace [38] (see page 59 for a description of PageSpace).

T Spaces

T Spaces [113], is a Java-based coordination middleware from IBM Research division, targeting a broad range of software architectures, from small embedded systems to large-scale distributed systems. Interaction with tuple spaces is done in the same way as Jada, monolithic tuple space servers are accessed from remote Java-based clients. A server contains one tuple space, but creating multiple tuple spaces on the same host is possible by running several instances of the server applications. A client wishing to access a tuple space server must specify its host address and port in order to perform tuple space operations. As for Jada, the way T Spaces manages communications is not suited to decentralised ad hoc environments because network members cannot speculate on the presence of other members and are not supposed to know every possible host/port couple at startup. Using such systems to develop decentralised ad hoc environment would require to build context management features on top of it. T Spaces tuples, classes extending the abstract `SuperTuple` class, are sequences of potentially named fields which can be any Java object implementing the `Serializable` interface. T Spaces primitives are extensible, users can define new primitives and download them to the tuple space server. Built-in primitives are the standard Linda primitives, to which are added the aggregate primitives (multiple `rd` and `in`) and the blocking *rendezvous* primitive (`rhonda`), which takes a tuple and a template as arguments, and succeeds when another client performs another `rhonda` operation with a matching tuple/template pair.

[1] An `ObjectSpace` is the implementation of a tuple space.

The standard matching process in T Spaces is as follows: a template and a tuple match if the tuple type is a subtype of the template type, the tuple and the template have the same number of fields, each tuple field is an instance of the type of the corresponding template, and, for each non-formal[2] field of the template, the field value matches the value of the corresponding tuple field.

Another way of accessing the tuple spaces is to use *queries* enabling matching operations to be combined using `or` and `and` operators. This helps to perform SQL's Select-like operations.

T Spaces provides mechanisms ensuring consistency in the tuple space server for both intra-operation and inter-operation consistency. The first is ensured by a checkpoint/recovery mechanism while the latter is ensured by a transaction system as in relational databases.

Security in T Spaces is achieved through an authentication process based on a simple login/-password mechanism and by access control at the tuple space level (each tuple space operation is defined with a set of attributes that an entity must possess in order to be allowed to execute the operation). However, C. Bryce, in [16], details flaws of T Spaces security mechanisms. It does not implement the concept of a reference monitor (see Section 3.2.3) which is prone to problems due to the lack of total mediation in open environments. Moreover, the cryptography applied to authentication and authorisation in T Spaces supports only very basic features [16].

T Spaces is actually reflecting too much its designers' background in database systems. They consider tuple spaces as databases and therefore implement mechanisms to deal with its consistency by using complex transaction systems. Indeed, this is useful to search for large amounts of data using complex queries, but it can be the source of low performances in systems where there are only a few tuples. Therefore, we consider T Spaces more suited to large scale centralised systems rather than to decentralised open systems in general.

JavaSpaces

JavaSpaces [48], is part of Sun's Jini [111] framework which allows interconnections of all sorts of devices that run Java virtual machines. JavaSpaces aims at providing distributed repositories of information, similar to Linda tuple spaces, to ease the design and implementation of distributed applications.

JavaSpaces are tuple spaces containing tuples represented as Java objects implementing the `Entry` interface. Thus, any object implementing it can be stored in a JavaSpace. Access to a space is specified in the `JavaSpace` interface. This interface allows to take, read and write `Entry` objects in the JavaSpace. Moreover, it allows to notify agents previously registered when an entry matching a given template is written in the space.

Each tuple is associated with a lease time, i.e. the tuple's lifetime. When it is over, the tuple is removed from the space. This feature associated with the notification feature can be useful to represent SMEPP events, as registered agents could be warned that an event has been produced and as no longer valid events would be automatically removed.

As in T Spaces, JavaSpaces allows to define transactions using external transaction services, for example the one supplied by the Jini technology.

JavaSpaces matching is static and is specified as follows: a tuple template which is an instance of class `A` matches a tuple which is either an instance of class `A`, or an instance of a class `B` which is a subclass of `A` if their fields match. Field matching is determined by the serialised form of the fields, being a byte array representation of the object. Two fields match if their serialised forms match. Linda matching mechanism is thus extended with subtype matching and field type matching.

JavaSpaces does not really tackle security. Any agent that owns a reference to a space can write, read or remove tuples from this space. For this reason, JavaSpaces is not yet ready to support secure applications [16].

JavaSpaces provides a mechanism to look up for newly created JavaSpaces by using the so-called `LookUp` service relying on the Jini lookup service. Entities wishing to publish a JavaSpace

[2] A formal field specifies a data type (e.g. string, integer, etc.), while an actual (i.e. non-formal) field contains an exact value (e.g. "abcd", 10).

must first obtain a proxy to this central `LookUp` service in order to publish it. Clients wishing to discover a JavaSpace have to contact the `LookUp` service. It is responsible to provide them JavaSpace references matching a template. This template specifies the properties the clients are looking for.

The `LookUp` service can be compared to the CORBA Naming Service as it provides a way to register and discover distributed objects [53]. Consequently, context management in SMEPP could be implemented on top of the JavaSpaces LookUp service. However, this would induce the difficulties presented in Section 3.1.1 regarding context management using CORBA Naming Service (i.e. the centralised implementation of the LookUp service and its unpredictable content due to the decentralised nature of SMEPP).

KLAIM

KLAIM [41], developed by R. Nicola, G.L. Ferrari and R. Pugliese, is a coordination model allowing agents to be moved from one computing environment to another. An available implementation of the model, named KLAVA [8], has been developed in Java. The KLAIM model is based on Linda but allows multiple tuple spaces and borrows operators from the process algebra CCS [67]. In order to achieve its mobility goal, KLAIM processes are *network-aware* and the network is composed of *locations*, considered both at logical and physical level, where processes can move and execute.

The main concepts used to model the network are *processes*, *nodes* and *nets*. *Processes* represent the active entities of the system and are located at a given *location*. *Nodes* represent the *locations* and are modelled as triples of the form (s,P,ρ), where **s** is a *site* (a physical location), **P** represents the set of processes of the node, and ρ is the allocation environment, mapping logical localities to their associated physical localities (*sites*). *Logical localities* are symbolic names for nodes while *physical localities* are their unique identifiers, i.e. their addresses among the *Nets*. *Nets* are sets of nodes and represent a logical view of the network.

KLAIM tuples and operations are located at specific sites of a net. All operations can be invoked by specifying a given location where the operation must be performed. For instance, `out(t)@l` writes the tuple `t` at the location `l`. KLAIM operations are the classic Linda operations: `in`, `inp`, `rd`, `rdp`, `out` and `eval`. In addition, a timeout can be specified when calling `in` or `rd` to specify the time within the operation must be accomplished.

Among *nets*, KLAIM provides a hierarchical structure for the *nodes*. *Nets* contains *subnets* for which they act as gateways, they route the information of the *subnets* members if necessary. For instance, let node A and node B be in the same *subnet*. If A wants to send a message to B, A must forward its message to the gateway which then forwards the message to B. One can see that this system does not suit decentralised architectures because of its hierarchical topology. However, KLAIM allows another communication system, allowing a process to connect directly to another one. The resulting communication is unidirectional. Therefore, there is no built-in support for peer-to-peer bidirectional communication. Still, this feature can be programmed by the application developer, but requires additional programming effort [7].

KLAIM deals with security using static access rights mechanisms enforced by the "network administrator". Access rights must be pre-defined at the agent startup. They restrict which tuple spaces it can access and which operations it can perform. The granularity of this access control mechanism is not fine enough to specify which tuples an agent is authorised to produce, read or retrieve.

SecOS

In contrast to the models presented above, the SecOS model, developed by C. Bryce, M. Cremonini and J. Vitek, uses a dynamic security policy [17]. This means that the reference monitor described in Section 3.2.3 exists at run-time and is integrated into the system [16]. The main purpose of SecOS is to provide coordination mechanisms between agents in mistrusted environments.

The key concept of SecOS is that keys are associated with objects *in* tuples. An agent must provide a matching key in order to access the corresponding tuple object. SecOS allows to use

either symmetric or asymmetric keys. In the symmetric case, the same key used to lock the object is used to unlock it. The same key is then shared by all the processes allowed to actually unlock the object. In the asymmetric case, key pairs are used. When one key of the pair is used to lock an object, the other is used to unlock the object. As we explained in Section 3.2.3 about the PAP protocol, one of the key is kept private while the corresponding key is public to every process.

SecOS is implemented in Java [18]. It reuses security mechanisms introduced by JavaSpaces: the reference monitor does not allow the matching process to be redefined and only copies of objects are stored within the space.

SecOS provides two ways for implementing the locking mechanism. Within a Java environment, objects and key pairs are protected using Java typing. A specific type exists and can only be interpreted as *access right*. No other data can be considered as an *access right*. Typing safeguards against access rights fabrication and makes rights propagation easier to detect and control [77]. However, this mechanism cannot be used when entities do not trust each other. They question their mutual ability to produce trusted access rights. Type safety is also not feasible when data are sent over the network, stored on disk or when tuples are stored in spaces of untrusted environments [16]. To solve this problem, when an object and a key pair *K:v* are used outside a trusted environment, an encryption key is associated with the key object K, encrypting the object v. In this way, the object is never exposed in a untrusted environment. Keys distribution is done by using the tuple space. Keys are stored in it and must be encrypted with other keys in order to protect them.

Using SecOS, the PAP implementation proposed in Section 3.2.3 can be easily implemented as it provides all the abstract mechanisms required. Therefore, any agent in SecOS can act as a PAP to enforce an application security policy using asymmetric keys. However, the problem of this security mechanism in the context of SMEPP is that it does not embody decentralised systems as explained in Section 3.2.3 (PAP implementation). Moreover, SecOS does not provide built-in mechanisms to deal with context management and mobility.

LindaCap

LindaCap [106] is an abstract run-time system implementing a coordination model developed by A.Wood and N.I. Udzir. This model aims at proposing finer access control mechanisms than those based on access control lists [24] or capabilities [17, 41, 43, 52, 56, 60, 71, 84]. A capability can be seen as a "ticket" granted to an agent by the system. This ticket represents the binding between an object and the agent's right on it. It has been established that capabilities offer more flexibility than access control lists [30, 95, 112], an important requirement for systems where users can join and leave at will. Capabilities are held by the agents and can be transfered in an agent-to-agent way. Therefore, using capabilities, an incoming agent wishing to access an object simply has to request a capability from the other members of the system. When using access control lists, numerous lists attached to the required object have to be modified and maintained, making the process less flexible. The problem of capabilities is that they must refer to single named objects. In the context of Linda, only tuple spaces are single named object, while tuples are not. Therefore, capabilities for tuples cannot be defined. The LindaCap model intends to provide a solution to this problem by introducing the concept of *multicapabilities*.

Multicapabilities allow to define rights bound to a set of *unnamed* objects, such as tuples. In addition to this feature, multicapabilities allow agents to create private partitions of the tuple space based on a set of tuple. This set is specified by a standard Linda template. For instance, let two agents `A` and `B` interact with a tuple space `ts` in order to work with templates containing two fields: a string and an integer. Before being allowed to perform an operation concerning the set of tuples specified by this template, `A` and `B` must first obtain a multicapability for it. `A`'s multicapability consists in a unique identifier α, the template and a set of permissions. Permissions can be either `i, r, o` representing the rights to perform `in, rd` and `out` operations related to the above mentioned template. `B`'s multicapability for the same template will be identified by a different identifier, i.e. β. Let `A` perform `ts.out('a',1)` using its multicapability. If `B` performs `ts.in('a',1)`, no matching tuple will be found and the operation will block. The reason is that a multicapability forms a private partition based on a template for an agent.

To allow `B` to perform operations on `A`'s tuple, `A` must share the multicapability α with `B`. Therefore, agents are the only actors deciding to grant access to their partitions, providing a decentralised access control management. The way multicapabilities are shared and managed will be explained later.

The LindaCap model allows to define both multicapabilities for templates and unicapabilities (standard capabilities) for tuple spaces. Unicapabilities contain two parts: the tuple space concerned and the set of operations allowed on it by the capability holder. The structure of the model is based on the following rules:

- Every tuple space and every tuple operation require a capability. Each agent performing an operation on a tuple must obtain a unicapability bound to the tuple space where the tuple is stored (or where it will be written). In addition, a multicapability for a template matching this tuple is required. Therefore, the unicapability for a tuple space and the multicapability for a template are passed as arguments of every tuple space operation (`in`, `rd` and `out`).
- Every request for a new capability returns a new and unique capability (even for identical templates), with full rights. An agent can request a new full-right multicapability for a template *tmp* by calling the `newcap(tmp)` method.
- An agent can create a tuple space by calling the `TupleSpaceC()` method. This method returns a full-right unicapability to the agent.
- The system provides a universal tuple space (UTS) which exists independently of the agents. It is publicly accessible as every agent possesses a default unicapability with full rights on it. However, they must acquire multicapabilities for specific templates before performing tuple operations on the UTS.
- Every agent owns a default multicapability allowing to write or read *capability tuples*. Every agent can read any *capability tuple* by using the template `?cap` as argument of a `read` operation. This multicapability allows agents to share and discover multicapabilities. Indeed, thanks to this multicapability, an agent can share one of its acquired capabilities by sending it in the UTS. Then, other agents will be able to read this capability by performing an `rd(⟨?cap⟩)` operation on the UTS.
- An agent owning a multicapability allowing operations on a template can produce from it other *lower privilege* multicapabilities. These capabilities still allow to operate on the same templates, but do not allow as many operations to be executed. This mechanism can, for instance, be useful to define read-only tuples. An agent acquires a full-right multicapability for a specific template, and then only publishes in the UTS a multicapability allowing read operations.

The advantages of LindaCap in the context of SMEPP are the following: multicapabilities provide a way to define dynamic access control mechanisms in a decentralised way. The agents are responsible of managing and sharing the access rights to their tuples and tuple spaces. Therefore LindaCap's security model can be used to implement a higher level of security than SMEPP's default level. LindaCap also provides means to deal with context management issues by providing a default multicapability and the UTS tuple space. Once an agent joins the network, it can use the default multicapability to discover capabilities which could represent groups or services information. However, the main problem of LindaCap regarding SMEPP requirements is that it does not embody the ideas of peer-to-peer systems. Indeed, the creators of tuples, tuple spaces and capabilities have a specific role. All the above mentioned objects are not publicly "shared" but created by one entity allowing or not access to them. Therefore entities cannot be considered as peers. Moreover, in case of an object's creator death, LindaCap proposes the garbage collection of the created objects. The consequence is that when a tuple space creator gets disconnected, LindaCap system could decide to delete the tuple space if necessary[3]. This mechanism is extremely problematic in the context of SMEPP as a group

[3]For instance, when the memory usage is higher than a certain percentage.

creator's death does not result in the deletion of the group. Therefore, LindaCap cannot be used as it is to implement the SMEPP middleware. However, its security model is an interesting approach to deal with dynamic access control in a decentralised way.

SecSpaces

SecSpaces [52] is a coordination model, developed by N. Busi et al., supporting secure coordination in open environments. SecSpaces proposes access control mechanisms at the tuple level, meaning that the control information is stored within tuples. This way of implementing access control is inspired by the SecOS model [17]. In SecOS, a `read` operation is a succession of an `in` operation followed by an `out` operation [17]. There are two consequences of this design. The first is that there is no distinction between a destructive and a non-destructive operation. Therefore, it is not possible to model an information which can only be read by specific agents, and not retrieved. In SMEPP, this feature is interesting as the middleware must ensure that an information published by a peer cannot be deleted by other peers. The second consequence is that an agent can reproduce any tuple it previously read. Any reader automatically receives producer permissions. This is problematic in the context of the SMEPP middleware. Indeed, event publication can be modelled by a tuple sent from a producer to a tuple space. Therefore, subscriber can discover the event by reading the tuple. The problem is actually that subscribers automatically become event producers by reading the event tuple.

The SecSpaces model intends to solve these problems by introducing two field-based security mechanisms: *partition fields* and *asymmetric partition fields*. Each of these two kinds of fields has two variants: one used in case of a `read` operation, and the other used in case of an `in` operation.

A *Partition field* provides a way to partition the tuple space by modifying the matching rule. A template and a tuple match if the template provides the exact value of the partition field of the tuple, and if the other fields of the template match the other fields of the tuple. Therefore, a *partition field* can be seen as a symmetric key used to allow the agents knowing it to access a partition of the tuple space. The main issue concerning *partition fields* is their distribution. The implementation must decide whether or not distributing them as tuples in a tuple space. If the system allows them to be distributed as tuples, a mechanism must be found to ensure that they cannot be discovered by untrusted agents. Indeed, an agent wishing to share a partition field within a tuple space certainly would like to share it with only specific trusted agents.

SecSpaces introduces *asymmetric partition fields* as a mechanism ensuring this property. Within a partition, an *asymmetric partition field* can be attached to a tuple. This field can take two specific values, being either `K` or `CO-K`. `K` is only known by the agents authorised to send the tuple to the tuple space, while `CO-K` is only known by the agents authorised to read or retrieve this tuple. Therefore, the matching rule must be modified in such a way that only the templates specifying `CO-K` match the tuples having their *asymmetric partition field* set to `K`, and symmetrically. The problem is that the middleware must be able to check the correspondence between `K` and `CO-K`, and that this information cannot be shared as tuples. Indeed, this would trigger the same problem highlighted by *partition fields* as they could be read by untrusted entities. Therefore, they must be static and private. Moreover, the implementation of the partition fields must be done in such a way that an agent knowing `K` should not be able to guess `CO-K` from it, and symmetrically.

An implementation of *asymmetric partition fields* matching rule has been proposed using asymmetric cryptography. An asymmetric partition field is implemented as a triple $(p, PubK, s)$, where p is a string, $PubK$ is an asymmetric public key and s is a string encrypted with an asymmetric key. If s corresponds to the string p encrypted with the key k, s can be written $\{p\}_k$. Using this representation, the matching rule is defined as follows:

- Let $PubK$ and $PubK'$ be the public key associated with the private keys $PrivK$ and $PrivK'$
- Let `K` $= (p, PubK, s)$ and `CO-K` $= (p', PubK', s')$

- `K` matches `CO-K` if $s = \{p'\}_{PrivK'}$ and $s' = \{p\}_{PrivK}$

The first remark regarding this implementation is that an agent knowing only `K` is not able to produce `CO-K`, and symmetrically. To produce `K`, an agent needs to know a private key $PrivK'$ and the public key $PubK$ corresponding to the private key $PrivK$ used in `CO-K`. As the knowledge of $PubK$ is not sufficient to produce $PrivK$, `CO-K` cannot be derived from `K`. Therefore, if we assume that an agent `A` keeps `K` private and that every other agent knows `CO-K`, it is impossible for a reader to reproduce a tuple protected by `K`[4].

The second remark regarding the implementation is that it provides a way to create secure communication channels between two agents. If an agent `A` wants to send a tuple to and only to an agent `B`, `A` must set `K` to $(p, PubK_B, \{p\}_{PrivK_A})$, and `B` must set `CO-K` to $(p, PubK_A, \{p\}_{PrivK_B})$. Thanks to this mechanism, even if an agent `C` knows the string p, it will not be able to produce a matching `CO-K` field as it does not know $PrivK_A$.

The last addition proposed by the SecSpaces security model is to define two predefined *partition fields*: one allowing `read` operations and the other allowing `in` operations. The same addition is available for *asymmetric partition fields*. This allows to define a finer grained access control policy discriminating between read and removal authorisations.

The SecSpaces security model provides mechanisms allowing to develop SMEPP default security level and security *level 2*. *Partition fields* can be used to model group communications, while *asymmetric partition fields* can be used to model access control policies. Moreover, *asymmetric partition fields* allow to authenticate agents and to create secure communication channels. The main issue is to find a decentralised way of distributing public keys within tuple spaces. In order to do it, we could assume the presence of a universal tuple space, such as LindaCap's UTS [106], where the SecSpaces kernel could store agents' identities. Having SecSpaces enhanced with the UTS mechanism would also provide context management features which are not foreseen by the current model.

SecSpaces is at the moment only a theoretical model without implementation. In addition, the model does not provide mechanisms to deal with context management issues. Therefore, it must be extended with such features and implemented in a decentralised way in order to be used to implement the SMEPP middleware.

Bach

Bach is a coordination model, developed by J-M. Jacquet and I. Linden, which has been extended in [60] to provide support for mobile ad hoc applications. We present this language incrementally by firstly explaining the core concepts of the bach model. Then, we present how these concepts can be extended to suit mobile ad hoc applications.

Bach's model conceive the network as a set of *hosts* identified by a name representing their physical location. Every host can house one or more *blackboards*, i.e. tuple spaces. Therefore, Bach provides remote communication between agents running on different hosts using *blackboards* as communication media. Agents communicate through *blackboards* using four timed primitives: (i) $tell_n(bbn, t)@l$ allows to send the tuple t to the blackboard *bbn* hosted in l for n units of time, (ii) $ask_n(bbn, t)@l$ allows to check the presence of the tuple t in the blackboard *bbn* hosted in l under the constraint that the operation fails if no result is returned within n units of time, (iii) $nask_n(bbn, t)@l$ checks the absence of the tuple t in the blackboard *bbn* hosted in l under the constraint that the operation fails if no result is returned within n units of time, and (iv) $get_n(bbn, t)@l$ removes the tuple t from the blackboard *bbn* hosted in l under the constraint that the operation fails if a matching tuple is not found within n units of time. In order to avoid clocks synchronisation problem between hosts, Bach assumes that the primitives refer to the clock associated with the host on which the targeted blackboard resides. In addition, Bach must ensure unique naming for hosts. This can be accomplished, as proposed in [60], by using MAC addresses as values for the l parameter.

Bach provides a way to ease remote blackboards access by allowing to define relations between them. These relations are used to assert the presence or absence of tuples on blackboards

[4] Excepted if the reader is `A`.

from the presence or absence of (possibly other) tuples on blackboards [60]. *Blackboards relations* take the following form:

$$in(b_1, t_1), ..., in(b_m, t_m), nin(b_{m+1}, t_{m+1}), ..., nin(b_n, t_n) \rightarrow$$
$$in(b_{n+1}, t_{n+1}), ..., in(b_p, t_p), nin(b_{p+1}, t_{p+1}), ..., nin(b_q, t_q)$$

This relation expresses that the presence of tuples $t_1, ..., t_m$ on blackboards $b_1, ..., b_m$ and the absence of tuples $t_{m+1}, ..., t_n$ on blackboards $b_{m+1}, ..., b_n$ implies the presence of tuples $t_{n+1}, ..., t_p$ on blackboards $b_{n+1}, ..., b_p$ and the absence of tuples $t_{p+1}, ...t_q$ on blackboards $b_{p+1}, ..., b_q$. A variable can be used instead of a tuple t to express the conditions "any tuple present in a blackboard" and "any tuple absent from a blackboard". Moreover, a tuple having certain fields set to variables can be used to match patterns of tuples.

Two semantics are possible for a *blackboard relation*: *forward reading* and *backward reading*. *Backward reading* states that the presence of $t_k (n + 1 \leq k \leq p)$ can be deduced from the presence of $t_1, ..., t_m$ on $b_1, ..., b_m$ and the absence of $t_{m+1}, ..., t_n$ on $b_{m+1}, ..., b_n$. The same condition determines the absence of $t_l (p + 1 \leq l \leq q)$. *Forward reading* states that when a new tuple $t_i (1 \leq i \leq m)$ is inserted in b_i, if the left-hand side of the relation is satisfied, the right-hand side tuples must be appropriately created or deleted. The same process must be performed when a tuple $t_i (m + 1 \leq i \leq n)$ is removed from b_i. The reader noticed that the evaluation of $in(b, t)$ or $nin(b, t)$ can result in the deletion of tuples from b. If the removal is not wished, a non-destructive version of *in* and *nin* can be used by writing $[in]$ and $[nin]$.

When writing a *blackboard relation*, the programmer can explicitly specify whether he wants to use the relation as a *forward reading* relation or a *backward reading* relation. The former requires to write $\rightarrow_f$ while the latter requires to write $\rightarrow_b$. For instance,

$$in(b_1, \langle Namur, 12, X \rangle) \rightarrow_b in(b_2, \langle Namur, 12, X \rangle)$$

states that the presence of any tuple of the form $\langle Namur, 12, \_ \rangle$ in b_2 is sufficient to deduce its presence in b_1.

To ease remote communication, Bach predefines on each host a local blackboard *bbv*, called the *default blackboard*, which contains two default *blackboard relations*. Whenever an agent wants to communicate with a remote blackboard *bbn*, its request is firstly sent to *bbv*, which will forward it to *bbn* using the first default blackboard relation. The second reaction non-destructively forwards any tuple sent to *bbn* in *bbv*. Therefore, these two default reactions create the illusion of a local *bbn* for the agent. It does not have to know anything regarding *bbn*'s location.

In the original Bach model, *Blackboards relations* are stored as special tuples within a centralised tuple space, because relations can concern blackboards on multiple host. Because of a lack of centralisation and because of connections instability in ad hoc networks, this centralised implementation could not be used in ad hoc context. Therefore, Bach's adaptation for ad hoc networks restricts reactions to those concerning only one blackboard. By restricting relations to one blackboard, they can be stored and managed locally by the host housing the concerned blackboard[5].

As many *blackboard relations* can be defined on the same tuples, Bach provides priority mechanisms to define their order of execution. Priorities are represented as an integer ranged from 0 to 255 and are attached to a *blackboard relation*. By default, a relation is created with the maximum priority, being 255. Bach executes relations from the highest priority one to the lowest. In case of equality, an arbitrary choice is made by the system.

Bach's extension for ad hoc applications comprises reaction mechanisms in response to events. Observable events in Bach are the connection of a host with a host h, the disconnection of a host from a host h, and the change of connection quality within a host h. In response to these events, reactions can trigger specific actions. For instance, it is reasonable that upon connection, a host offers a forward relation from its blackboards to the blackboards of the connected host. This can be simply programmed by linking the *default blackboard* of the old host with the *default blackboard* of the new host through a *forward blackboard relation*. This reaction can be written as follows:

[5]The interested reader can find more details on the restrictions in [60].

$$in(\langle connected, X\rangle) \Rightarrow tell([in(Y)@self] \rightarrow_f [in(Y)@X])@X$$

where the $\Rightarrow$ arrow indicates that the forward reaction is activated as soon as a tuple matching ($\langle connected, X\rangle$) is found, where $self$ refers to the name of the existing host and where X represents the name of the newly connected host. Reactions can be extended by specifying tuples on their left-hand side and a sequence of actions on their right-hand side.

Bach's extension for ad hoc applications also provides access control mechanisms. Capabilities are associated with each tuple and with each blackboard primitive. These capabilities take the form of hidden attributes of tuples, blackboard relations and primitives. These attributes are inherited from the process which has led to the creation of the considered tuple, the considered blackboard relation and the execution of the considered primitive [60]. For instance, connection event tuples can only be sent by system processes. Any agent which cannot provide capabilities corresponding to system processes is not able to delete this kind of tuple. Another example is that a forward relation from X to Y requested by Y should be allowed by X. To ensure this, Y's capability is bound to the blackboard relation. If this capability matches X's one, the forward relation is authorised. This last example can be useful in the context of SMEPP, to only allow peers having the right credentials to use blackboards relations. These blackboard relations could for instance model group communication.

As Bach provides context management and security mechanisms in a decentralised way, it is a good candidate to implement the SMEPP middleware. However, it is at the moment only a theoritical model. Moreover, the process of defining all the blackboards relations and the process of allocating capabilities is a complex task in the context of SMEPP. In addition, the way of allocating, creating and managing capabilities is not yet specified in [60].

3.3.2 Systems adding programmability

LuCe, ReSpecT and TuCSoN

`LuCe` (Logic Tuples `Ce`ntre) is a coordination model, developed by E. Denti and A. Omicini, for the construction of multi-agent systems involving autonomous, pro-active and possibly heterogeneous agents [45]. An implementation of the model, the LuCe system [42], has been developed in Java over a lightweight Prolog engine. It allows to develop both Java and Prolog agents over a web environment. The key contribution of the model is the concept of *tuple centre*, a *programmable coordination medium* [43]. The behaviour of a tuple centre can be defined by using specific tuples called *specification tuples*. A tuple centre is then composed of two parts: the *tuple space*, containing ordinary *communication tuples*, and the *specification space*, containing specification tuples. The state of the *tuple space* provides a *communication* viewpoint while the state of the *specification space* provides a *coordination* viewpoint. Indeed, the behaviour of the tuple centre governs inter-agent communication, and specification tuples define the rules of inter-agent coordination. Therefore, agents have two levels of action, reasoning on the communication viewpoint and the coordination viewpoint, by possibly refining or changing the coordination laws.

The feature of programming tuple centres behaviour allows to uncouple the actual representation of knowledge in a tuple centre from the agents' perception of it [93]. Tuple centres can be programmed to bridge the different representation of information shared by agents. They are suitable to support the full *monitoring* of agents interaction and to house the laws for agents coordination [44]. Taking these features into account, the tuple centre could be compared to the notion of *reference monitor* we presented in Section 3.2.3, encapsulating and enforcing security protocols such as authentication and authorisation.

The behaviour specification of a LuCe tuple centre is defined through the ReSpecT specification language [44], a logic-based language where *reactions* in response to communication events are defined by means of first-order logic tuples, called *specification tuples* [93]. Observable events are: a tuple sent, a tuple read, a tuple retrieved and no tuple matching a template found. Additional information on the events can be obtained, such as: which agent triggered the event, which tuple is involved in the event, which operation triggered the event, in which tuple centre the event occurred and information about success or failure of an operation (pre,

post, success, failure). Reactions can also produce tuple centre operations such as `out`, `in` and `rd`, which can trigger a new reaction. Thus, chained reactions are allowed by the model. We could therefore imagine an authorisation process that only allows a specific agent to retrieve a specific tuple containing the agent's permissions. However, in the context of SMEPP, it is impossible to define entirely the behaviour of the tuple centre in order to grant such static authorisations to all possible peers, as their number and identities are not known in advance. In addition, the *specification space* is observable and alterable by agents. There appear to be no mechanisms to secure it, making it unreliable in untrusted environments. Regarding mobility concerns, LuCe does not provide mechanisms to ease context management in open and decentralised environment.

An interesting feature of Luce is however to allow services to know the state of the tuple centre. An agent can know the set of its tuples, the set of its pending queries and the set of its reaction specifications. We can feel that LuCe targets more the development of autonomous and decision taking agent societies rather than the development of decentralised and potentially untrusted applications requiring mobility and security features.

TuCSoN [84], developed by A. Omicini and F. Zambonelli, is a coordination language highly inspired by LuCe, from which it borrows the concept of tuple centre and the ReSpecT specification language. The main difference between both models is that TuCSoN provides a hierarchical infrastructure. Each tuple centre is associated with a *node* which can act as a gateway for the other nodes of which it is responsible. Gateways are connected to each other in a tree way. This enables, thanks to the ReSpecT specification language, to define nested protected domains, each one controlled by a specific access control policy. This feature is interesting in the context of groups and sub-groups. As SMEPP does not define the concept of sub-groups and, as a hierarchical infrastructure is not suited to the decentralised peer-to-peer orientation of SMEPP, TuCSoN's architecture appear less suitable than LuCe's architecture. The last feature we want to highlight is how the communication is achieved using TuCSoN or LuCe. Using LuCe, tuple centres are only accessed in a *network-transparent* way, meaning that entities do not have to know any detail about the physical locations of other entities. In TuCSoN, the communication can be either *network-transparent* or *network-aware*.

Law-Governed Linda

Law-Governed Linda (LGL, [71]), developed by N.H. Minsky and J. Leichter, is an instance of the concept of *law-governed architecture* [70], which defines interactions governed by a global law specified in the architecture. LGL includes the five basic components of the law-governed architecture:

- **Communication medium.** This is the means used by system actors to communicate. LGL defines the Linda tuple space as its communication medium.
- **Sequential processes.** Processes (or agents) are the system actors.
- **Control states.** It represents the information about processes and their history. Each process has its own control state that is used to know the history of the accesses to the communication medium.
- **Global law.** This unique law, composed of a set of *rules*, provides a way to govern the interactions taking place in the communication medium. The rules are written in Prolog and are of the form: `on` *event* `if` *conditions* `do` *actions*. A complete specification of the law language can be found in [74]. This language can be compared to the ReSpecT specification language, but adds more events and the possibility of taking context into account by using control states.
- **Law enforcement mechanism.** A *Controller* is associated with every process, acting as a mediator between the process and the communication medium. Its role is to enforce the global law. As every process is associated with its own controller, the enforcement mechanism is fully *decentralised*. Every permission is authorised or refused by the controller in charge of a process.

As in LuCe, each access to the communication medium by an agent generates an *event* which is caught by its associated controller. The controller searches the global law for a rule which describes the occurred event and the actions to be taken in terms of primitive operations. This process is called the *ruling* and is influenced by the process control state. The primitive operations that the *ruling* can trigger are as follows. It can execute the operation (i.e. allowing the operation), send a tuple to the tuple space[6], remove a process from the system and allow the tuple matching a template to be returned to the calling process.

LGL intends to provide a way to address the problem of peer-to-peer coordination within a group of agents being governed by the global law. The remark we made regarding the absence of built-in protocols for authentication and authorisation in LuCe also holds for LGL. However, several kinds of security policies can be implemented by exploiting the programmability of the law rules [72].

An extension of LGL, *Law-Governed Interaction* (LGI) [74, 73, 69], implemented in the Moses middleware [74], aims at providing a coordination mechanism that allows open group of distributed active entities to interact with each other. The interaction is ruled by an explicitly specified security policy, called the law of the group. Groups are considered open as members can join and leave different groups at will, being governed by different laws. Because of its relevance to enforce security policies among different groups and because it provides a way to program peer-to-peer communication, LGI might be a good candidate to implement the SMEPP proof-of-concept. However, choosing this solution to implement SMEPP enforces us to define a finite set of groups and thus, a finite set of laws. Indeed, the creator of the group in SMEPP does not have more privilege than the other members and cannot specify himself the rules applicable for its group. Furthermore, we cannot consider a central entity fixing dynamically new rules and new groups. Therefore, the problem is to pre-define all the possible groups and their associated rules.

We consider that the main problem to use LGI and other law governed systems is to deal with authorisations in a unpredictable environment. In such systems, all the possible users cannot be known in advance. Therefore, it is not possible (and not wanted in the case of SMEPP) to define authorisation rules for each of them. A solution to this problem could be to gather users into predefined categories for which laws and rules are established. In this case, the user which does not match these predefined categories would receive fixed default rights.

Another problem is to determine how to solve mobility issues when using LGI as it does not explicitly provide means to solve the issues of context management and unpredicted disconnections. The Prolog-law language offers however elements to tackle these issues as *creation*, *disconnection* and *reconnection* of entities are observable events. It is not specified whether unpredicted disconnections are observed, but if they are, on disconnection, the controller could trigger the deletion of the disconnected entity's tuples. This would solve the problem of space consistency in case of unpredicted disconnections, but the cost of this solution in terms of tuple space accesses and messages sent would be high. There is also the need to ensure that none of the being-deleted-tuples would be accessed by other agents during deletion. This last point seems difficult to express using the Prolog-like language.

MARS

MARS (Mobile Agent Reactive Spaces)[24], developed by G. Cabri, L. Leonardi and F. Zambonelli, is a coordination architecture which supports *programmable reactive tuples spaces* for Java-based mobile agent applications. Mobility in MARS concerns agents roaming the network in a *network-aware* way. The network is modelled as a *set of sites* referenced by their physical location. Each site can be seen as an execution environment, a host, for a finite number of nodes. This way of considering mobility is not easing the task of modelling the way SMEPP considers it, as peers do not know in advance the locations of other peers. The MARS model assumes that there exists one specific unnamed tuple space in each site. This tuple space is *local*, i.e. independent of the other space sites. It is the only mean for agents to communicate locally on a site or remotely with other sites. When an agent is created on a site, the

[6] Which will not be considered as an event.

system provides it with the reference to this local tuple space. The agent can therefore start to communicate with other agents by using it. The choice of defining a local tuple space is a fundamental property for mobile systems. It helps to overcome the resource binding problems involved by the change of the execution environment [49] and leads to the implementation of the *context-dependent interaction* concept. However, this concept is related to code mobility, and not to physical mobility: it resolves problems related to the migration of an agent code on a site and not problems related to the unpredicted disconnection and reconnection of entities.

The access to the tuple space is highly inspired by JavaSpaces. It is implemented in Java with an interface extending the `JavaSpace` interface. MARS introduces two new methods to the `JavaSpace` interface: `readAll` and `takeAll`. The former provides a way to read *all* the tuples matching a template. It helps to solve Linda's multiple read problem, that is, Linda's inability of reading each matching tuple exactly once [93]. The latter allows to retrieve all the tuples matching a template.

As LuCe and LGL, MARS tuple spaces are *programmable* and *reactive*. They can be programmed in response to access events. The difference with the other models is that a MARS reaction can influence the semantics of the access operations. Indeed, MARS allows the effects of the existing primitives to be changed. For instance, a reaction may change the pattern matching mechanism to let agents specify a range of value rather than a fixed value in the template of an input operation [25]. Reactions are implemented using *meta-tuples*, stored in a local *meta-tuple space*. When an access event occurs, the system searches a meta-tuple matching it. When such a tuple is found, the system calls a method of this tuple modeling the reaction. In order to avoid endless recursions, MARS does not allow reactions to be triggered by other reactions. MARS meta tuples can be stored and retrieved at run time, allowing dynamic management of reactions. These tuples can be manipulated by agents and by the "local administrator".

MARS assumes the existence of a local administrator whose role is to define the reactions, access control lists and roles. Security in MARS can be addressed by using these three concepts. As in LuCe and LGL, reactions can implement a specific security policy. MARS goes further than LuCe and LGL by allowing the administrator to define *access control lists*(ACL) for each tuple. It models *who* can access tuples and *what* operations can be done on them. *Roles* can be defined by the administrator to ease the management of ACL. Agent's identities can be directly mapped into roles defining the access rights associated with them. For instance, roles can be *reader*, *writer* and *manager*. A reader can only read tuples, a writer can read tuples and store their own tuples but cannot retrieve tuples produced by other agents. A manager (or local administrator) can perform any operation on the tuple space, modify access control lists and manage reactions by modifying the meta tuple space.

A Java implementation of the MARS model has been developed [23]. The implementation is made in such a way that it wraps existing mobile agent systems. In order to perform the integration of the agent system, the implementation defines an *agent server* component, in charge of accepting and running new agents on a node. This component also provides new agents with the local tuple space reference and acts as an interface to perform tuple space operations. The implementation has been tested with the following agent system: Aglets [59], Java-to-go [102] and SOMA [80].

3.3.3 Systems modifying the model

PageSpace

The PageSpace platform [38], developed by P. Ciancarini, A. Knoche, R. Tolksdorf and F. Vitali, is a meta-architecture or a reference architecture for developing Internet-based applications [33]. This architecture relies on the notion of agents using a coordination technology to interact. The coordination technology explored by the developers of the architecture are Linda-like coordination languages. This is why we classify PageSpace as a tuple space based coordination model while it is actually an architecture where the coordination technology can vary. The coordination technology, or coordination architecture, can be seen as the operating

environment of PageSpace. This is a shared workspace where agents live and communicate. Its nature influences the way agents are created and the way they interact.

To understand the PageSpace architecture, it is important to make clear the notions of applications, agents and users. Applications are composed of a set of independent agents interacting with each other using the PageSpace (i.e. a shared data space). This inter agent communication defines the applications behaviour. Users own a *homeagent* providing them the PageSpace reference in order to communicate. Users are using WWW browsers in order to communicate with the PageSpace. They are able to start or stop agents and to react in response to agents' messages. The PageSpace architecture provides a framework for supporting multi-user applications and inter-application communication. It also offers support for changes in the configuration of users and applications. Users logging into the PageSpace might use unreliable and/or non-persistent connections, leading to unpredicted disconnections. This situation should not interrupt the regular functioning of the application. Therefore, PageSpace provides ways to allow the disappearance and reappearance of the users. Thus, it does not only tackle code mobility problems, but also physical mobility ones. This results in the fact that PageSpace provides mechanisms to deal with context management, which is an important feature required for the SMEPP middleware.

To understand how the system actually works, we present an overview of the different agents running in the coordination architecture:

- *Homeagents* are a persistent representation of users. As users can be absent or disconnected, *homeagents* collect messages addressed to their user, queue them in order and deliver them on user request. They also can act on messages and requests from other agents when the users are absent. Thus, every agent wishing to communicate with a user addresses its message to the user persistent *homeagent*. Only this agent "knows" its user presence information.

- *Application agents* are the agents performing the operations required by the application specification. They can be started and interrupted by users. They can also communicate with the shared data space and provide services to other agents. Some application agents do interact with users and provide a *user-interface agent* in the form of a HTML document, a Java applet, etc. This interface is reachable from the user browser and is the only way to interact with the application. Other applications do not need to be controlled by users and therefore do not provide interface agents. Such application agents just communicate and offer services to other agents.

- *Gateway agents* provide access to the external world for PageSpace applications. Applications needing to communicate with other environments send requests to the appropriate gateway agents, translating them to the destination environment. The gateway agent is also responsible for translating and returning the corresponding response to the agent. Each gateway agent is responsible for one specific destination environment.

- *Kernel agents* perform management and control tasks. The main components of the kernel agents are:

  - the *agent store*, responsible of storing the application agents' programs. Each application is mapped as a process within the kernel. This process executes the operations and manages the exceptions occurring during the computation.
  - the *homeagent server*, which stores homeagents and ensures their persistence.
  - the *Repository* component, which models the shared data space and the operations allowed on it.
  - the *agent connectors*, which enable multi-agent communication on the same data space.
  - the *kernel connector*, which allows for communications between distributed kernels.

  Kernel agents also manage agents/services discovery and naming. At agent or service creation, meta information is written by the kernel agents in the form of a web page

accessible through HTTP. Other users and agents can then discover newly created services or users by querying this page maintained by the *kernel agents* in the form of a search engine.

Implementations of PageSpace have been done using Jada [36] and Laura [105], which is a tuple space language easing the development of service-oriented applications. However, these implementations are not available regarding the project web page [6]. Moreover, the limitations of PageSpace regarding the SMEPP proof-of-concept implementation are that it does not specify any security mechanism and that the available documentation does not specify whether the kernel implementation suits decentralised applications management. However, looking at the *homeagents*, it appears that persistent servers should remain alive even if no users use the system.

Lime

Lime (Linda In a Mobile Environment) [75], is a coordination model developed by L. Murphy, G.P. Picco and G.C. Roman, addressing the issues of code and device mobility in shared data spaces environments. Lime revisits the tuple space model in order to adapt its advantages of time and space uncoupling to a mobile environment.

The key concept of Lime is the way tuple spaces are shared among mobile entities, leading to the concept of *transiently shared tuple space*. Each mobile entity (either an agent or a mobile device) is associated with an *Interface Tuple Space* (ITS). This tuple space is the personal space of the entity, where tuples can be stored and retrieved from. An entity can have one or more named ITSs, identified by a distinct name. When entities are physically connected, their ITSs having the same names are merged. Thus, both ITSs are seen by the entities as a unique tuple space containing the tuples of the two entities' ITSs. When an entity moves, the Lime system recomputes the transiently shared tuple space for that location, taking into account the tuple spaces of the incoming entity. This process is called the *engagement*. The reverse process is called *disengagement*. When an agent leaves the location, the *disengagement* consists in the removal of the leaving agent's tuples from the transiently shared tuple space. A mobile entity can choose to keep one of its ITSs private by not engaging it. This provides a way to keep private information.

To illustrate the *engagement* and *disengagement* processes, let two agents `A` and `B` connected with each other, both having an ITS named `ts`. If `A` performs an `out(t)` operation on `ts`, then, if `B` performs an `in(t)` operation on `ts`, `t` is retrieved by `B` because both ITSs merged.

Using the same example, if `A` decides to move to a new location before `B` performs the `in(t)` operation, the tuple `t` is removed from the transiently shared tuple space. Therefore, when `B` performs `in(t)`, no matching tuple is found and `B`'s execution is interrupted. In order to avoid this situation, `A` can send the tuple `t` to `B`'s ITS by performing an `out[B](t)` operation instead of the classical `out` operation. In this case, even if `A` migrates, `B` is still able to retrieve the tuple `t` as it is stored in its own ITS instead of `A`'s ITS.

In order to allow `out[B](t)` to be performed, Lime adds two specific fields to each tuple, being their current location and destination location. The current location specifies which agent is currently storing the tuple, while the destination location specifies the agent to which the tuple will eventually be moved. If we assume that the agent `A` performs `out[B](t)`, the execution is done in two steps. Firstly, the tuple `t` is sent to `A`'s ITS, having its current location set to `A` and its destination location set to `B`. If `B` is connected, the tuple is moved by the system to `B`'s ITS and its current location is set to `B`. If `B` is disconnected, the tuple remains in `A`'s ITS until `B` connects. In this case, the tuple is considered *misplaced* as its current location and its destination location differ.

An interesting feature of Lime is the ability to program reactions in response to events. One can program such reactions using the `T.reactsTo(s,p)` method, called *reactive statement*, where `s` is the reaction code to be executed when a tuple matching the template `p` is found in

the tuple space T. Thus, events in Lime only take into account tuples' insertion. Lime provides a second *reactive statement* allowing to unregister a reaction from a tuple space. Reactions are executed in the following way: after every *non-reactive statement*, one of the registered reaction is selected arbitrarily by the system. If the condition to trigger it is satisfied, the reaction is executed. Otherwise, the system selects non-deterministically another reaction. This selection and execution process stops when no more reactions can be triggered. In addition, Lime's reactions can be used in response to configuration change in the system. This is achieved by using a default tuple space called the *LimeSystem* tuple space.

Every entity creates at startup its local version of the *LimeSystem* tuple space. Therefore, whenever entities are connected with each other, the *LimeSystem* tuple space is always merged. This tuple space helps to solve the issue of knowing which entities are currently in a particular location or to know which entities belong to a specific set of connected peers. Withdrawal and insertion of tuples in the *LimeSystem* tuple space is restricted to the run-time system. However, every entity can read this tuple space to gather system configuration information or to register reactions in response of configuration changes in the system.

Lime suits really well the development of mobile agents communities. Context management issues are tackled by two of Lime's features. On the one hand, the engagement and disengagement process ensures consistency of the tuple spaces in case of agents' arrivals or departures. On the other hand, the *LimeSystem* tuple space provides a way for agents to discover other agents by reading its content or registering reactions. Therefore, agents do not have to know anything concerning other possible agents of the system. They even do not specifically have to know which agents is using the system as the run-time system automatically merges tuple spaces having the same name.

An open source implementation of Lime [88] has been made in Java and is well documented. The features of Lime seem really relevant to ease the development of groups in the context of SMEPP. Two other variants of Lime have been implemented: TinyLime [40] and TeenyLime [39]. Both variants are adaptation of the Lime model for mobile devices. The former is an adaptation of Lime for sensor networks, while the latter is designed for sensor and actuator networks.

In addition to the features previously mentioned, Lime provides a tool for simulating a GPS system. This tool is started along with every Lime application. It provides a fake GPS location to the application and allows the programmer to move it across the space. This tool enables to simulate an application disconnection due to its departure from the network and an application reconnection due its arrival. This is very useful to simulate complex scenarios and to test space consistency after unpredicted disconnections.

The main limitation of Lime is that it does not provide built-in security mechanisms except the notion of private tuple space. This is not sufficient to model the notion of visibility of groups and services in SMEPP. There must be a way to avoid specific agents to discover, join and participate in groups to which they do not have access. However, as the implementation is done in Java, the Java cryptography library [101] can be used to encrypt tuples with specific keys related to the visibility of the agent.

MESHmdl

The MESHmdl middleware [57], developed by K. Herrmann, specifically aims to coordinate applications in ad hoc settings. To deal with the dynamics of ad hoc networks, MESHmdl introduces the concept of *mobility aware* applications. Such applications rely on a middleware providing them information to *self-adapt* according to changes in their *neighbourhood* (i.e. the devices within connection range). In the following, we present the design of the MESHmdl middleware which supports the development of the aforementioned *mobility aware* applications.

MESHmdl is based on two key concepts: mobile agents and the tuple space model. On the one hand, mobile agents are used because they can adapt to the dynamics of connectivity by moving to a different location. Indeed, when a communication is known to take some time, the invoker can move to the location of the provider to avoid disconnections due to connectivity

problems. On the other hand, the tuple space model is used as it introduces a high degree of uncoupling. Indeed, entities wishing to communicate do not have to exist in the same location at the same time.

The architecture of the MESHmdl middleware is composed of six layers:

- the *agent application* layer is the top layer of the middleware. It contains all the running applications (i.e. the active mobile agents).

- the *agent runtime* layer manages the running mobile agents. It consists in an *Engine* component, running on every node, which hosts mobile agents and manages their migrations.

- the *space* layer represents the tuple space. Every *Engine* runs a unique tuple space which is the only way for agents to communicate with each other. An agent can communicate with its local tuple space, but also with the neighbouring tuple spaces (i.e. the tuple spaces managed by nodes in connection range). Note that the communication between an agent and its *Engine* is also performed via the tuple space. Thus, the programmer is provided with a uniform interface for all communication tasks [57].

- the *interaction layer* provides a way for agents to discover their neighbourhood. In contrast with Lime [75], this is not achieved in a transparent way, but in a *network-aware* one. Every *Engine* is associated with a *node entry* which identifies it. The *interaction layer* is in charge of sharing the *node entries* of the *Engines* in connection range. This is achieved by an engagement protocol. When two nodes enter in connection range, the engagement protocol is run and the nodes exchange their *node entries*. If the protocol successes, each node stores the node entry of its new neighbour in its local tuple space. When a node disconnects, the engagement protocol is run to delete the corresponding entry from one's local tuple space. One can see that this protocol is similar to the *blackboard relation* created in response to a connection event in the Bach coordination language (see Section 3.3.1, p. 55).

  Node entries are the key elements enabling remote communication. They act as a handle to gain access to a neighbouring node [57]. When an agent wants to communicate with one of its neighbour, it can specify the corresponding *node entry* as an argument to a method named *go*. This method (handled by the *Engine*) migrates the agent to the targeted neighbour where it can continue its execution. Please note that the migration is performed through the tuple space. The agent's data is wrapped inside a tuple written in the targeted agent's space. Upon arrival the tuple is retrieved, unwrapped and its content is executed.

- the *generic connection layer* enforces the notion of neighbourhood and encapsulates the *network layer*. It presents the aforementioned layers with neighbour nodes and primitives for connecting to them [57].

- the *network layer* represents the networking technology (e.g Bluetooth, Wifi, etc.).

An interesting feature of the MESHmdl middleware in the context of SMEPP is the *Xector* model. This model eases information diffusion in ad hoc environments. A Xector is managed by the *Engine* and consists of a node template n, a data template d, a specifier being either *collect*, $inject$ or $reject$ and a filter f which manipulates the entries before information diffusion. For instance, a Xector $< inject, d, n, f >$ is executed on a node X as follows: when a node whose node entry matches n arrives in X's neighbourhood, $inject$ (put) every of X's tuples matching d into the space of the new neighbour. Before the actual transmission, each entry is filtered by f. In general, $inject$ implements a push strategy while *collect* implement a get strategy. The $reject$ mode allows to reject the insertion or the collection of tuples in one's tuple space. Again, the Xector model shows similar features to the ones provided by the programming of *blackboard relations* in response to connection events with the Bach coordination language (see Section 3.3.1 page 55).

MESHmdl appears as an interesting candidate to implement our SMEPP middleware proof-of-concept because of its relevance in ad hoc environments. Indeed, mobile agents combined with the engagement protocol and the Xector model offer adaptation mechanisms in response to environmental changes. However, MESHmdl does not provide enough security mechanisms to model the SMEPP security model. Indeed, the *reject* mode of a Xector only allows to reject entries pushed by specific agents and to reject entry requests performed by specific agents. The problem is that SMEPP peers do not know in advance all the possible peers. Therefore, they have no ways to statically specify a *reject* Xector for specific peers. They only can statically define a Xector rejecting every peer.

Still, a *reject* Xector could be used to implement SMEPP group security if group members were identified by a specific *node identifier*. Indeed, a group member could specify a Xector concerning the group tuples which would reject every agent having its *node identifier* different from the group one. However, the specification of MESHmdl seems to allocate such identifiers for nodes, and not for agents. Moreover, in SMEPP, nodes belonging to different groups can be executed on the same host. This complexes the attribution of *node identifiers*.

While *reject* Xectors do not appear relevant to model SMEPP group security, they offer an interesting way to define dynamic exclusion policies. Indeed, a *reject* Xector can be registered by a peer to exclude another peer identified as a maliciously behaving one.

In conclusion, the MESHmdl middleware cannot be used as it is to implement our SMEPP proof-of-concept but offers an interesting framework to build ad hoc applications.

SecureLime

SecureLime [55, 56] is a coordination system, implemented by R. Handorean and G.C. Roman, extending the Lime coordination model. It extends Lime with security features. Lime's advantages to model mobile agents communities are therefore preserved in SecureLime. The addition to the original model intends to provide secure sharing of tuples and tuple spaces.

In Lime, the *LimeSystem* tuple space can be read by any agent. This tuple space contains all the names of the existing tuple spaces, agents and hosts. While this tuple space can only be read-only accessed, an agent can read the name of a tuple space. Therefore it can create an instance of this tuple space which will be merged with the existing one in case of physical proximity. Then, the agent gets full access to it.

To solve this issue, SecureLime introduces the concept of *password protected tuple spaces*: in addition to its name, a password is associated with the tuple space. In Lime, the name of the tuple space is the key required to access it. Therefore, in SecureLime, the associated password is used to avoid that the fact of reading a tuple from the *LimeSystem* tuple space allows full access on it. This feature is implemented in the following way: at tuple space creation, an agent can decide to associate a password with it. If no password is chosen, the tuple space is accessible by every agent. If a password is chosen, it is used to encrypt the name of the tuple representing the tuple space in the *LimeSystem* tuple space. Therefore, when an agent reads this tuple and extracts the name, if it creates a tuple space with this encrypted name, it will not be merged with the actual tuple space. In order to be merged, the name of the tuple space must be the one found in the *LimeSystem* tuple space decrypted with the corresponding password. The advantage of this security feature is that it is fully decentralised: no central entity is used to check passwords or to grant access to a tuple space.

In the default SMEPP security level, the credentials used to access groups are pre-shared and static. Therefore *password protected tuple spaces* could be used to model SMEPP secure groups as the only fact of knowing a password would grant access to the corresponding group.

SecureLime proposes another security mechanism: access control at the tuple level. Whenever agents have access to a secured tuple space, they have full access to it. Therefore, agents could perform denial-of-service attacks by retrieving all the tuples of the tuple space. However, we can assume that this situation will not happen because the users of the tuple space can be considered trusted as they own its password. While this attack is unlikely to take place, another problem might occur at the tuple level because of Lime's polymorphic matching rule.

Every tuple field is a Java `Object`. Therefore, if an `in` operation with a template containing two `Object`s is performed, every two-fields tuple will be retrieved. Agents performing this operation might not be interested in every tuple, and are not likely to send them back in the local tuple space they were retrieved from. The information represented by the retrieved tuples could be permanently lost as the owner of these tuples would not be warned of their removal.

SecureLime provides two mechanisms to avoid this situation: *read-only tuples* and *password-based matching*. The former allows the owner of a tuple (the agent who sent it to the tuple space) to declare it *read-only*, meaning that it cannot be retrieved by any other agent. The owner location is added to the tuple in order to implement this feature. When an agent intends to perform an `in` operation on a *read-only* tuple, its location and the owner's location are compared. The operation is allowed only if both locations are the same. In the case the owner decides to move this tuple to another location, SecureLime updates the location of the tuple.

The latter allows to read or retrieve a tuple according to the knowledge of a password associated with the tuple. This is achieved by defining two password fields, one for remove protection and one for read protection, which cannot be matched using the standard polymorphic matching rules. In order to match, the agent performing an `in` or `rd` operation must specify the exact value of the password field corresponding to the operation. The other tuple's fields are not affected by a change of the matching rule and can then be matched in a polymorphic way. In addition to the polymorphic and exact value matching, SecureLime provides the *exact type* matching rule. This rule allows to use types in patterns' fields[7]. Every tuple field of that type will therefore match. This matching rule differs from the polymorphic one in such a way that it does not allow wildcards[8].

As these passwords are tuple fields, and as tuples can be remotely accessed, SecureLime provides a way to protect tuples being sent across unsafe networks. A malicious entity could catch on the network a packet corresponding to a tuple. Then, it could discover the tuple's content and the password fields. This process is called eavesdropping. Two mechanisms are used to protect against eavesdropping. The first one is that, when sent over the network, tuples belonging to a secure tuple space are encrypted with the tuple space password. In this way, if a malicious entity eavesdrops this tuple, its content will not be readable if the agent does not know the tuple space password. If the malicious entity knows the tuple space password, it can decrypt the tuple's content, but SecureLime avoids it to discover the password fields. In fact, these fields do not contain the real values of the passwords. They contain a static predefined string `''secretText''` encrypted with the actual passwords values. Therefore, an entity which does not know the actual passwords cannot discover them by eavesdropping the tuple over the network.

SecureLime security mechanisms appear to be an interesting way to deal with authorisations in a decentralised way. The only flaw is that eavesdropping the content of a tuple is possible for entities belonging to the same group even if password fields have been set. However, this limitation is not constraining regarding the SMEPP security requirements because they assume peers belonging to a group to be well-behaved. Moreover, performing this attack is complex. The attacker must first capture a packet representing a password protected tuple. Identifying this packet is a difficult task as the data is in raw byte form. Then, the attacker must translate the byte representation in an Object representation. The final step is to decrypt the tuple using its corresponding tuple space password. Deriving the tuple space containing the caught tuple from its byte form is extremely complex as the tuple does not contain this information. Therefore, the attacker must know the origin of the tuple only by analyzing its byte form.

3.4 Choice of the target language

This section presents the choice of the tuple space based coordination language on top of which we shall implement the SMEPP proof-of-concept, taking into account the description of the

[7] These fields are called formal fields.

[8] In SecureLime, wildcards are formal fields of type `Object`.

languages made in Section 3.3 and the SMEPP middleware requirements presented in Section 2.5.

A result of the tuple space based coordination systems survey made in the previous section is that while there exist numerous extensions of the Linda model, few of them address context management in ad hoc networking. Surveyed languages, such as MARS, KLAIM and LGL present the notion of mobile agents roaming the network in a network-aware way, meaning that information about the physical location of the entities is a requirement to communicate. The problem is that these languages do not provide ways to manage unpredicted configuration changes in ad hoc environments such as SMEPP. Moreover, they do not introduce mechanisms to discover physical locations of the entities of the network, making context management difficult to implement. The surveyed languages providing ways to deal with context management are LGI, PageSpace, Lime[9], LindaCap, Bach and MESHmdl.

LGI provides ways to monitor agents' connections and disconnections by offering means to react to these events. The problem is that it is not specified whether unpredicted disconnections are caught by the event monitor. Moreover, the task of ensuring space consistency after disconnection must be implemented from scratch, in contrast to PageSpace, Lime and SecureLime.

The PageSpace model provides mechanisms, such as the *homeagent*, to deal with the disconnection and reconnection of entities. The main problem of this system is that its implementation does not seem available. The project page does not state the existence of any implementation while the paper describing the project relates an existing one. Moreover, few details on the run-time systems are available. Still, the features of the system tend to induce the notion of centralised servers dealing with presence information. This kind of run-time systems can obviously not be used in ad hoc environments.

LindaCap provides context management mechanisms through the default multicapability present in the universal tuple space (UTS). This multicapability allows any new entity to search for capabilities, which could help it to discover new tuple spaces and new tuples' multicapabilities. Moreover, multicapabilities allow to define complex security policies which could be used to model SMEPP's security. However, while an entity defines global authorisations using multicapabilities, it seems difficult to define authorisations concerning only a specified set of entities. In addition, as explained in Section 3.3, LindaCap's model does not embody the notion of peer-to-peer system. Finally, there is currently no available implementation of the LindaCap model. Therefore, we cannot use this model to implement the SMEPP middleware.

Lime, in contrast, is explicitly designed to solve mobility issues in ad hoc environments. The concept of *transiently shared tuple space* eases the development of mobile agents communities and provides an interesting solution to tackle context management issues.

Bach, as Lime, tackles context management issues in a decentralised way. This is provided by *blackboard relations*, by the default *bbv* blackboard easing remote communication and by the reaction mechanism. Indeed, when a host connects to an existing host, a relation can be defined by the existing host to forward its blackboards to the new host. All the aforementioned mechanisms provide a way for entities to communicate with each other in a *network-transparent* way, i.e. they do not have to know anything about physical locations. Bach, regarding reactions, is more expressive than Lime. Indeed, reactions cannot only be triggered by the presence of a tuple, but also by its absence. However, *blackboard relations* definition is a time-consuming and error prone task. Indeed, one must appropriately declare events sending relations in blackboards. Attention must be paid regarding non-destructive and destructive operations in *blackboards relations*. It is also important to ensure the consistency of a set of *blackboard relations*. In Lime, tuple spaces are *transiently shared* by the system, making the management task much easier.

MESHmdl, as Lime, is explicitly designed to solve mobility issues in ad hoc environments. Indeed, *mobile agents* combined to the *engagement process* and the *Xector* model offer interesting adaptation mechanisms in response to changes in the environment. However, in contrast with Lime, the MESHmdl middleware does not ease the implementation of group communication. In Lime, the *transiently shared* tuple spaces provide an intuitive way for modelling

[9] And its extension SecureLime.

SMEPP groups. In contrast, MESHmdl defines a unique tuple space for each node in which peers from different groups have to communicate. The problem is that we showed that MESHmdl security mechanisms do not permit to model SMEPP static group security (see Section 3.3.3 p. 64). Therefore, there is no way to avoid peers to discover information related to group to which they do not belong. Finally, the MESHmdl implementation, related in [57], is not available for download.

In conclusion, Lime and its extension SecureLime appear as the best languages according to the peer-to-peer orientation requirement (`R1` in Section 2.5) of the SMEPP middleware. Moreover, both languages satisfy the Java-integrability (`R4` in Section 2.5) requirement and the available implementation requirement (`R3` in Section 2.5).

Regarding the security requirement (`R2` in Section 2.5), SecureLime appears to be an interesting solution to model group security using *password protected tuple spaces*. SMEPP default security level requires the use of pre-shared symmetric keys. These keys could be modelled by tuple space passwords while groups would be modelled by the tuple spaces themselves. Moreover, SecureLime provides a way to secure communications in a untrusted environment.

The limitations of SecureLime security policy are the following. Firstly, malicious group members are able to eavesdrop tuples from a group tuple space. However, this limitation is not an issue regarding SMEPP security requirements as group members are supposed to be well behaved.

Another limitation is that SecureLime only provides symmetric cryptography mechanisms at the tuple space and tuple level. Therefore, SecureLime can only be used as a coordination technology to implement the SMEPP default security level.

SecOS and SecSpaces, in contrast, provide asymmetric cryptography mechanisms at the tuple level, which could be used to implement the highest level of security envisioned by the SMEPP middleware. The main problem with SecOS and SecSpaces is that these languages do not conceive the network as a mobile network and therefore do not provide means to implement context management features.

The last security limitation of SecureLime we want to highlight is that the granularity of its access control mechanisms is lower than the other surveyed languages. It only allows read and remove protections, while KLAIM, LuCe, TuCSoN, LGL, LindaCap, Bach and MARS allow to define access control mechanisms for every tuple space operations.

In conclusion, SecureLime is the only surveyed language fitting mobile ad hoc environments, offering decentralised security mechanisms and being implemented in Java. However, regarding security, a number of the surveyed languages offer more possibilities. Still, SecureLime security mechanisms appear expressive enough to implement the SMEPP default security level. We therefore have chosen SecureLime to implement our SMEPP middleware proof-of-concept.

Table 3.4 summarises the features of each surveyed language relevant in the context of SMEPP. Namely, these features are: the requirements of Section 2.5, support for timed primitives, and support for event creation. While the requirements of Section 2.5 have been previously detailed, the last two are additional features interesting in the context of SMEPP. On the one hand, timed primitives allow to model tuples lifetime. They also provide a timed alternative to read and remove operations. This avoids to use probe operations when blocking operations cannot be used because of application requirements. On the other hand, the coordination system's ability to model events and their associated reactions provides an efficient way to model reactive applications.

3.5 SecureLime API

Since our implementation, which is presented in the next chapter, is built on top of the SecureLime API, it is necessary to give details about it. This section intends to make the reader comfortable with the API by presenting its concepts used in the proof-of-concept implementation. Note that, since SecureLime is an extension of Lime, an important part of the following

Languages	R1 P2P	R2 Security	R3 Implementation	R4 Java	Timed	Events
Jada	no	yes	yes	yes	yes	no
T Spaces	no	yes	yes	yes	no	no
JavaSpaces	no	no	yes	yes	yes	yes
KLAIM	no	yes	yes	yes	yes	yes
SecOS	no	yes	yes	yes	no	no
LindaCap	no	yes	no	no	no	no
SecSpaces	no	yes	no	no	no	no
Bach	yes	yes	no	no	yes	yes
Luce	no	yes	yes	yes	no	yes
TuCSoN	no	yes	yes	yes	no	yes
LGL	no	yes	no	no	no	yes
LGI	-	yes	yes	yes	no	yes
MARS	no	yes	yes	yes	yes	yes
PageSpace	-	no	no	no	no	no
MESHmdl	yes	no	-	yes	no	no
Lime	yes	no	yes	yes	no	yes
SecureLime	yes	yes	yes	yes	no	yes

Table 3.1: Summary of the surveyed languages' features.

applies also to the Lime API[10].

Lime Agent

A Lime agent is a Java-based program using the Lime API. Any Lime agent must implement the `ILimeAgent` interface to access tuple spaces. Lime offers two classes implementing it: `StationaryAgent` and `MobileAgent`. We only use the former. The latter provides code mobility features, which is not used in the context of SMEPP. Every Lime agent has a "manager" which, notably, holds an identifier for the agent. This identifier is in the form of an `AgentID` object. `AgentID`s are used to compute `AgentLocation`s which are used as location parameters in tuple space operations.

Lime Reaction

As explained before, Lime offers a system of reactions which triggers the execution of a code when a tuple matching a specified template is found. It is implemented through the `Reaction` interface.

The API offers two classes implementing `Reaction`: `LocalizedReaction` and `UbiquitousReaction`. The former not only requires the tuple space to contain a matching tuple, but also requires this tuple to match the current and destination locations[11] specified at the reaction's creation time. The latter specifies location-independent reaction. The reaction is thus installed on the whole tuple space.

Reactions are added to a `LimeTupleSpace`[12] by calling `addStrongReaction(LocalizedReaction[])` or `addWeakReaction(Reaction[])` on it. The first method registers a group of reactions to a tuple space. "The operation is performed atomically, i.e., even if some of the reactions become suddenly enabled by the current state of the tuple space, none can fire until all of them have been registered" [2]. This method requires the reactions to have locations fields specified. The second method registers also a group of reaction to a tuple space. In this case, "the operation is NOT performed atomically, as this would involve a distributed transaction"[2]. This method allows to specify reactions without location parameters.

[10] This section is based on [56], [75] and the SecureLime API documentation [2].
[11] Note that the destination location is optional.
[12] Or `SecureLimeTupleSpace`.

Any `Reaction` object is defined by a template (used to match the tuple), a reaction listener (containing the code to be executed) and a reaction mode. The mode is specified by two constants. On the one hand, the *ONCE* mode fires the reaction only once. On the other hand, *ONCEPERTUPLE* fires the reaction every time a (different) matching tuple is found.

The `ReactionListener` interface defines a method (`reactsTo(ReactionEvent)`) which contains the code to be executed. The method's parameter is used to receive the information related to an event that fired the reaction (i.e. the tuple that triggered the reaction, the associated reaction listener and the agent owning the tuple space where the event occurred).

System tuple space

The `LimeSystemTupleSpace` is a special tuple space that all Lime agents possess. Agents can only *read* this tuple space (no `in()` or `out()` operations allowed). The tuple space contains tuples giving information about the environment. When a new entity (host, agent or tuple space) is created, a tuple is added to the `LimeSystemTupleSpace`. When the entity is deleted (or disconnected), an "anti-tuple"replaces the corresponding tuple. Whenever the entity returns, its anti-tuple is removed and a new tuple is added.

- A tuple of the form <`_host,LimeServerID`> means a new Lime host (i.e. a computer) is connected to the network,
- A tuple of the form <`_agent,AgentID,LimeServerID`> means a new agent (i.e. a program) is connected,
- A tuple of the form <`_tuplespace,String,AgentID`>, where `String` is the name of tuple space, is created when a new tuple space is created.

Note that for each tuple, its "anti-tuple" has the same format but the tags are `_host_gone`, `_agent_gone` and `_tuplespace_gone`. This gives the programmer the ability to add reactions associated with "system" events.

Remember that in the case of SecureLime, the tuple space names in the `LimeSystemTupleSpace` are encrypted.

Probe operations

Lime offers non-blocking version of `in()` and `rd()`. It is important to signal that the `inp()` and `rdp()` operations are more restrictive than their original versions. Indeed, the API requires the current location parameter to be specified while using these methods. Thus, it is impossible to invoke those operations on the whole federated tuple space.

Aggregate operations

The aggregate operations are `outg([])`, `ing([])` and `rdg([])`. They extend their original versions by inserting or retrieving several tuples at a time. As in the case of the probe operation, these require the current location parameter to be specified. Thus, one cannot use these operations on the whole federated tuple space.

Note that the SecureLime API does not provide a secure version of `outg()`. Thus, one cannot insert several read and/or remove protected tuples at a time.

Protected tuple movement

SecureLime adds a specific restriction to tuple movements. It is forbidden for an agent to insert a protected tuple into another agent's tuple space. Indeed, the other agent may not have the corresponding password to remove the tuple.

Chapter 4

Proof-of-concept implementation

This chapter presents the SMEPP proof-of-concept implementation. This implementation consists in two main components: the *SMEPP middleware API* and the *Translator*. The *SMEPP middleware API*'s roles are to provide a SecureLime based implementation of the SMEPP primitives and a Java implementation of the SMoL commands, while the *Translator*'s role is to translate a SMoL specification to an executable Java application. Figure 4.1 gives a high level view of the proof-of-concept design. Moreover, it shows the steps to be achieved in order to translate a SMoL specification to an executable Java application.

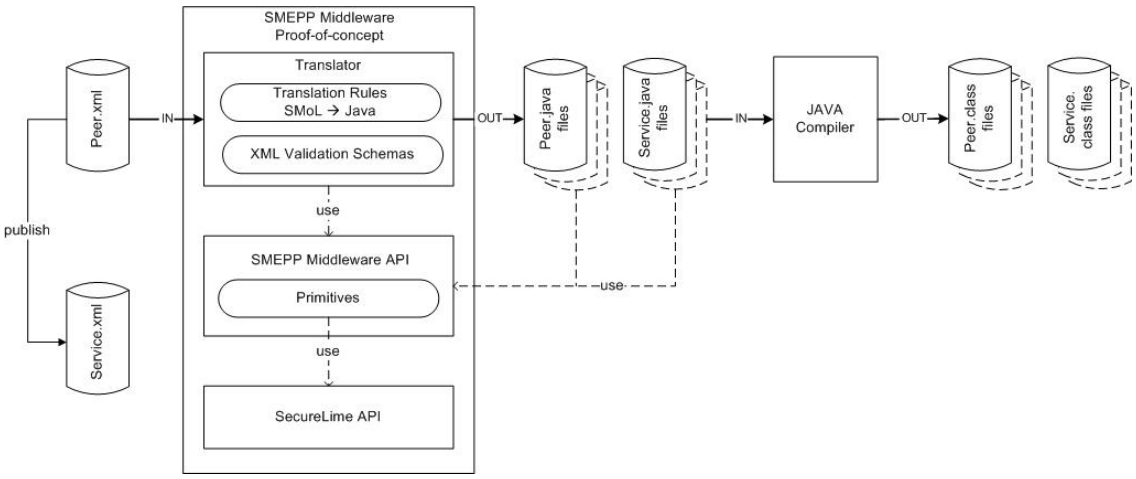


Figure 4.1: High level view of the proof-of-concept design

The SMoL specifications of peers and services are represented by XML files validated by XML schemas. These schemas specify syntactical constraints to be satisfied by the XML files.

In order to produce an executable application from a SMoL peer specification, one must firstly provide its corresponding XML file to the SMEPP proof-of-concept middleware. Then, the *Translator* checks its validity regarding the *XML Validation Schemas*. If this peer intends to publish services, the translator automatically checks their validity. In case of success, it produces the corresponding Java code. This process relies on *Translation Rules* and on the *SMEPP middleware API*. Indeed, *Translation Rules* are used to define the translation process for every SMoL commands, while the *SMEPP middleware API* methods are called from the produced Java code. Finally, the peer and the service Java files must be compiled using a Java compiler in order to be executed.

In the rest of this chapter, Section 4.1 presents the *Translator*'s design and the issues related to the translation of SMoL specifications. Section 4.2 presents the design and the SecureLime based implementation of the *SMEPP middleware API*. Finally, Section 4.3 shows an example of the translation of a SMoL specification and its execution.

4.1 SMoL to Java translator

This section describes how the translation from a SMoL input file to a Java application is done. The translation is twofold. Firstly, a careful study of the SMoL specification is needed in order to produce a suitable design of the translator. This study is presented in Section 4.1.1. Secondly, a parser, which takes as input an XML file (representing a piece of SMoL code), has to be built. This parser generates the Java code from the input file. Section 4.1.2 highlights parsing difficulties caused by the SMoL concrete representation. Finally, Section 4.1.3 describes the link between the translator and the SMEPP middleware API.

4.1.1 Translator design

In order to design a suitable architecture for the translator, it is necessary to understand the issues that the software has to face. Those problems to be solved come from the limitations of the SecureLime API as well as from the specification of SMoL. We present them in three points.

- The SecureLime API is very restrictive concerning the threads[1] which can access a local tuple space (or Interface Tuple Space, in SecureLime terms). Indeed, to invoke an operation such as `in()` or `out()` on a tuple space, the caller thread has to be either the creator of the Java Object representing the tuple space or one of the creator's children. Furthermore, the creator has to implement the `ILimeAgent` Java interface. The possible children must implement the `LimeThread` interface. Note that this last interface provides the only way to program concurrent execution using the SecureLime API.
- The specification of SMoL in itself defines strict execution behaviours. Some constructs have special termination and/or synchronisation characteristics of which to be aware.
  - The `Flow` construct terminates when all its children (which are executed concurrently) are finished. This means that the statements following a `Flow` cannot be executed before all the `Flow` branches.
  - `InformationHandler`s execute many commands concurrently. The main command and each branch have to be modelled by a thread, as well as each command associated with a branch.
  - Each branch of a `Pick` has to be modelled by a thread too. Furthermore, since only one branch can be chosen, all of them have to be synchronised.
- SMoL defines the management of faults in a quite different manner than Java does. In SMoL, when a fault is thrown, all executions have to be stopped inside the closest `FaultHandler`. In the example of Figure 4.2, if a fault is raised inside one of the `While`s, the *two branches* of the `Flow` have to be stopped. Then, the command associated with `CatchAll()` is executed. Intuitively translated to Java, this piece of code would become a main thread (modelling the `FaultHandler` and the `Flow`) with two child threads, one for each `While`. However, in Java, when an exception is raised inside a thread, only this thread undergoes the consequences of the exception. The parent thread is not aware that an exception has been raised in one of its children. Thus, it cannot notify the fault to its other children.

Altogether, those problems restrict the architecture design. Indeed, a SMoL application is intrinsically multi-threaded. Keeping in mind SecureLime's restriction on tuple space access, this means that every thread has to implement some SecureLime interfaces. And, when a fault is raised in the code, several threads have to be stopped. Furthermore, because of the way Java handles exceptions, it is not possible to simply translate SMoL faults into Java exceptions.

One could think to translate SMoL into Java directly, using anonymous Java threads to model multi-threading. However, this is impossible since all threads wanting to access tuple

[1] In this chapter, a "thread" refers the Java thread class (i.e. an independent stream of instructions which can be executed concurrently with others).

```
1   FaultHandler
      Flow
        While true
          ... ⟨faults may be thrown here⟩ ...
5       End While
        While true
          ... ⟨faults may be thrown here⟩ ...
        End While
      End Flow
10
      catchAll()
        ⟨do something to recover the fault.⟩
    FaultHandler
```

Figure 4.2: Fault handling example.

spaces have to extend `LimeThread` and implement `ILimeAgent`. This is impossible with anonymous threads because they are not defined in an independent class. Moreover, this solution does not allow any mechanism to handle SMoL faults. A better solution would model all primitives and constructs as objects extending `LimeThread`. This allows each command to be executed concurrently and it permits to define methods in the objects to handle faults. Although this design seems elegant, it induces an important overhead at runtime. Indeed, for each single command, the creation of a Java thread is needed. Therefore, we have chosen a compromise between the direct translation and the all-objects view.

We firstly divide the SMoL commands in two categories. The commands involving multi-threading (i.e. `Pick`, `InformationHandler` and `Flow`) are translated to an object oriented view by using Java interfaces. Other commands (such as `wait`, `if-then-else`, etc.) are translated directly to Java. The `FaultHanlder`, while being a single-threaded command, is also translated to a Java object. This translation allows to represent nested SMoL constructs by linking each command to its parent. For instance, in Figure 4.2, the `FaultHandler` is the parent of the `Flow`. Moreover, all commands (here, the two `While`s) inside the `Flow` are its direct children. The modelling of a `FaultHander` as a Java object allows its children to use the `FaultHandler` reference to forward SMoL faults.

Note that SMoL makes the implicit assumption that a root `FaultHandler` is the ancestor of all commands. This `FaultHandler` is in charge of forwarding the faults to the environment.

In addition, each command has to know the SMEPP entity executing it (i.e. a peer, service or session). This information is needed by child commands since some primitives behave differently according as they are invoked by a service or a peer. Therefore, the root of a SMoL program is contained in a SMEPP entity object.

Figure 4.3 presents the different objects used as a framework to translate SMoL. The rest of this subsection gives details about those objects.

ThreadedCommand. The main purpose of this interface is to define a set of methods that SMoL constructs have to implement in order to link the commands with each other (with regards to faults forwarding). `ThreadedCommand` also allows the primitives to retrieve information about the SMEPP entity which executes them. Furthermore, each object used as a SMoL command extends the `LimeThread` object and implements the `ILimeAgent` interface. The former allows multi-threading while using SecureLime and the latter makes the objects considered as Lime Agents, so they can call tuple space related primitives.

The `ThreadedCommand` interface is implemented by every other translated command. The top of Figure 4.3 shows the methods which must be implemented by the classes implementing it. The first method (`getUpperCmd()`) returns the parent command of the current one. `GetContainer()` is used to know whether the caller is either a peer, a service or a session. The two following methods are in charge of the fault handling. `IsStopped()` allows an object implementing `ThreadedCommand` to test whether it has to stop its execution or not, in case a fault is raised in a sibling thread. To signal a fault to its parent, a child calls `forwardFault()`.

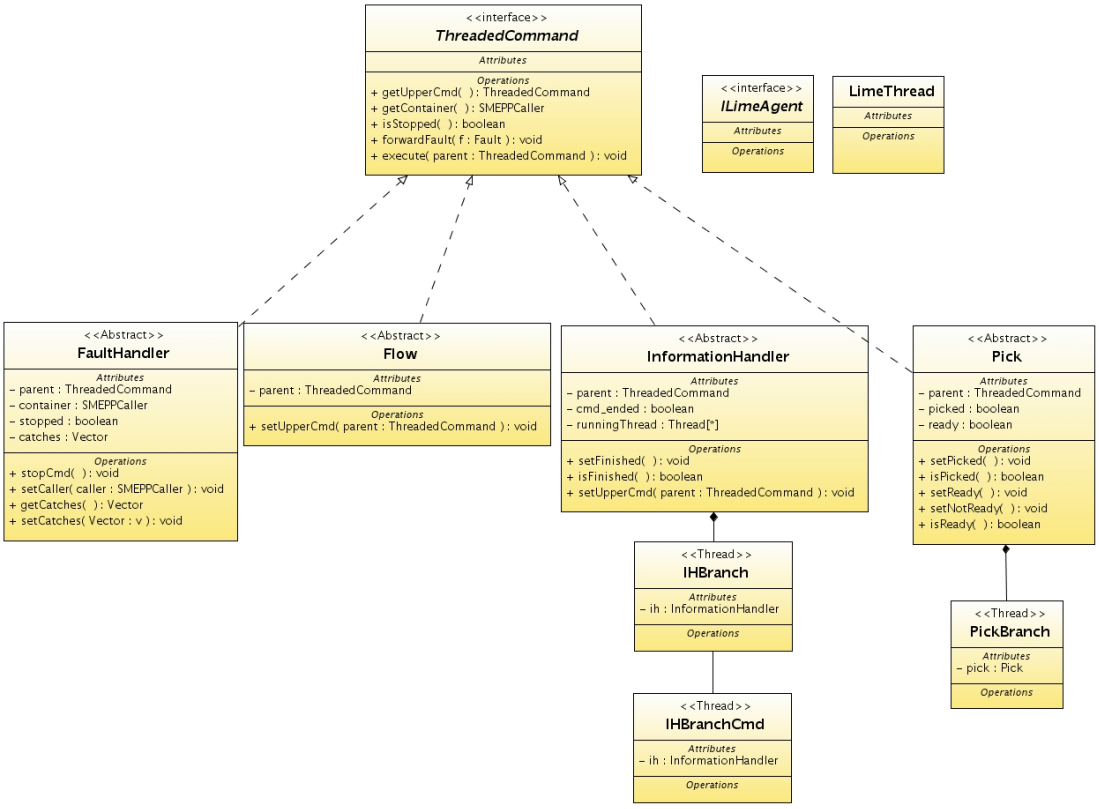


Figure 4.3: Translator's objects hierarchy.

Finally, the last method contains the actual body of the command.

FaultHander. This object is the only one which represents a single-threaded command. The `execute()` method simply contains the translation of its main command. The `FaultHandler` is the only construct which handles faults. It implements differently the `forwardFault()` method. Indeed, `forwardFault()` iterates on the list of `catch`es to find a corresponding command to execute. If no matching `catch` is found, then either the fault is forwarded to the parent or the program is stopped (in case it was the outermost `FaultHandler`).

In case of a service or a session, when the service is removed (i.e. unpublished), its execution has to be terminated. This is the role of `stopCmd()`, which is called by the service provider at the unpublishing time. Other methods are used to actually create the mechanism to match the `catches`.

Flow. The `Flow` object represents a branch of the corresponding SMoL construct. Every branch of a `Flow` is linked to the same parent (via `setUpperCmd()`). Thus, in case a fault is raised in a branch, the other children of a `Flow` can be aware of it by calling `isStopped()` on their parent.

InformationHandler. An `InformationHandler` contains three kinds of thread. The first one represents its main command which is executed on the main Java thread. Moreover, one thread is necessary for each branch (`IHBranch`). Indeed, each `receiveMessage()`, `receiveEvent()` and alarm needs to be executed concurrently. Finally, each time a branch is taken, a new thread executes the command associated with the branch (`IHBranchCmd`). Thus, we use three objects to translate an `Informationhandler`. The main one is used to synchronise the branches. Basically, the `cmd_ended` variable and the `isFinished()` method are used by the branches to know

when the main command is finished. In this way, they can terminate their execution as soon as the command terminates. The two other objects are simply Java Threads, which implement `ILimeAgent` and extend `LimeThread`. Moreover, they contain a reference to their parent to be able to call `isFinished()`.

Pick. The `Pick` is similar to the `InformationHandler` but requires more synchronisation mechanisms. In this case, one kind of thread is needed, one for each branch. However, all branches have to be synchronised with each other to assert that only one is taken. When a branch is "ready" to be taken, it signals it to the `Pick` object. The branch has then an "authorisation" to process. If its execution goes well (i.e. if it actually can execute the branch), it notifies its success via `setPicked()`. After that, the associated command can be executed and the `Pick` can terminate. More details on the synchronisation are available in Section 4.2.4 (see `receiveMessage()` and `receiveEvent()`).

As explained before, other SMoL constructs can be translated directly. A quick outline of their translation is presented in the following.

- **empty()** is simply translated to a semi-column,
- **exit()** exits the program, thus, it is translated to a call to `System.exit()` in Java,
- **wait(...)** is translated using the system call `Thread.sleep(...)`,
- **assign** is implemented using common assignation in Java (i.e. by using "`=`"),
- **if-then-else** is, obviously, translated to an `if-then-else`, and
- **while** and **repeatUntil** are translated to Java's `while`s.

Note that the `assign` command and the primitives handle variables. Thus, it requires to transform SMoL variables into Java ones. This is done as follows. Each SMEPP entity (peer or service) has a `DataTable` file associated. This file is a `static` Java class which contains all the variables of the corresponding SMoL file. Variables are translated to `static` attributes of the `DataTable` class. Accessing them amounts to access Java (`public`) attributes.

4.1.2 SMoL - concrete view

This section firstly presents an analysis of the SMoL concrete representation. The goal of this analysis is to show the representation limitations compared to the SMoL service model description. Then, we present a way to limit the impact of these limitations by implementing appropriate features into the translator. Finally, this section highlights an important gap between the concrete representation of SMoL and an executable representation.

Peers and services are written as XML files validated by two different XML schemas. Basically, the role of a schema is to define syntactical constraints that XML files must satisfy in order to be validated by it. Peers have to satisfy the constraints of the *SMoL.xsd* schema, available in Appendix A.2, while services have to satisfy those of the *Contract.xsd* schema, available in Appendix A.1.2. This schema refers to the *SMoL.xsd* schema, allowing to specify the behavior of the service and to the *Signature.xsd* schema, available in Appendix A.1.1. This last schema allows to define the signature of the operations proposed by the service. Please note that *SMoL.xsd* is actually inspired by the XML schema for BPEL processes [78] and that *Signature.xsd* is actually a schema borrowed from WSDL specifying web services [109]. The *Contract.xsd* schema has been developed by the SMEPP project to define service contracts corresponding to the service model description.

The SMoL schema defines XML elements representing every basic and structured command. Each of them is associated with a type which models the aforementioned syntactical constraints. For instance, `newPeer()`'s type, `tNewPeer`, expresses that when `newPeer()` is used, an input

parameter of type `tInputNewPeer` must be declared. In addition, `tNewPeer` states that an optional output parameter of type `tOutputNewPeer` can be declared.

The SMoL schema defines a central element, `Process`, of type `tProcess`. This element can be seen as the root of a SMoL-validated XML file. This element encapsulates all SMoL commands. Therefore, this root element allows to write programs which do not respect the service model description of SMoL. For instance, writing an XML file containing a `Process` element which encapsulates only a `catch` operation is considered valid. However, this is not semantically correct. A number of invalid programs which could be statically detected can be defined because of the `Process` element. Therefore, we implement validation mechanisms to avoid the execution of those programs. These mechanisms take the form of static checks made during the translation. For instance, one of them checks that no `catch` operation is defined outside a `FaultHandler`. Another one generates an implicit global `FaultHandler` if a program does not start with one.

In case an invalid situation is detected during the translation, an error is reported to the user and the translation is aborted. It is important to note that the current translator implementation does not ensure that every invalid program is statically detected. Actually, SMEPP considers that it is the programmer's responsibility to develop valid XML files. However, we implemented these static check mechanisms to ease the debugging of SMEPP applications.

The *Contract* schema defines a central element named `Contract`. Its type requires the following elements to be declared: `Signature` and `Grounding`. `Signature` refers to *Signature.xsd*, while `Grounding` represents implementation information concerning the service. In addition to these two required elements, `Contract` allows to define three optional elements: `Properties`, `QoS` and `Behavior`. `Behavior` refers to the SMoL schema and describes the SMoL behaviour of a service. `Properties` and `QoS` have not been considered in the scope of our prototype. Moreover, `Properties` appears to be redundant as it declares the same elements as `Signature`. It is important to note that the schemas have not been fully specified yet by the SMEPP project. Therefore, their structure is prone to change and some of their elements (such as `Properties`) have no definitive semantics yet.

This problem causes a gap between the current representation of SMoL and an executable representation. Indeed, various information have not been bound to an XML flag representing them. However, as we need to use these schemas in order to create executable programs, we have fixed the semantics of specific XML elements to model the needed information.

The aforementioned gap is actually enlarged when taking into account SMoL's XML representation of manipulated data. A SMoL specification is actually able of manipulating data. Indeed, as primitives are considered as simple commands in SMoL, their inputs and outputs must be considered as data which have to be manipulated by SMoL programs. The problem is that inputs and outputs of primitives cover a wide range of data types. For instance, primitives such as `invoke()` or `reply()` allow to define any object as inputs. The resulting issue is to express these data in XML and to ensure their translation into Java.

In the SMoL schema, there is currently no such data types definition. When studying any XML flag representing an operation manipulating data, we can see they are all represented in the same way. An operation input is represented by the type `tFrom`, while an operation output is represented by the type `tTo`. The type `tFrom` allows to declare the literal value of the input or to declare a variable containing the input value. The type `tTo` allows to specify a variable which will store the operation output. The issue resides in the way that literal values are represented by the XML schema. Actually, a literal value is an element of type `tLiteral`, which can encapsulate *any* element. Therefore, as every primitive input can be represented as a literal, a primitive can define anything as input, leading to invalid programs. For instance, one can declare `newPeer(3)` without generating a validation error. To avoid this problem, we implemented a set of data types in the SMoL schema. These data types represent the primitives input and output types. For instance, such data types are peers credentials, entities identifiers, etc. Therefore, we have a mean to check the type of an element contained in a literal.

When a primitive operation is translated, the translator checks that the literal value of input parameters are of the expected type. If not, an error is reported to the user and the translation

is aborted.

An issue regarding data types implementation is to define the types of the `invoke()` input parameter and the `reply()` output parameter. Indeed, they can take any object as values. To solve this problem, we implemented two specific data types: `Input` and `Output`, which allow to encapsulate any Java Object. The encapsulation is achieved by writing the `Object` constructor method within the `Input`'s or the `Output`'s `literal` element. Therefore, this is a limitation of our implementation in the sense that it introduces language specific information in the XML schema.

Another factor enlarging the gap is the `assign` operation. This operation allows to assign any data type (represented by a `tFrom type`) to any variable (represented by a `tTo` type). Therefore, the translator should be able to translate any data type to its corresponding Java form. Moreover, the XML representation of the data should have to be language independent, making the translation even more complex. This operation requires to scope every possible data types, to implement them with an XML type[2] and to foresee their translation into Java. In the context of this prototype, we decided to scope the translation of data because of the complexity of the previously mentioned tasks. Therefore, our prototype only translates into Java the data types used within input and output of primitive operations. When other data types have to be programmed, the programmer must manually modify the translated code. For instance, such code modification can be the casting of an `invoke` input parameter in its original data type. Indeed, we previously explained that the original input parameter is packed inside an `Input` type object. Therefore, when the `Input` object is retrieved, the programmer must cast it in its original type.

Other operations requiring manual code modifications are `while`, `if` and `repeatUntil`. All of these operations contain a `condition` element representing their associated boolean expression. The problem is that the type of the `condition` element does not specify which operations and which data can be considered when writing a boolean expression. A boolean expression can involve various data types and operations, not only the first order logic operations and the boolean data type. Indeed, a boolean expression can be $\texttt{myVariable}_1$ `+ 5` $\leq$ $\texttt{myVariable}_2$`.anyMethod()`. Such an expression refers to variable contents, to arithmetic operations and to method invocation. The task of representing complex boolean expressions in XML and to translate them in Java is not trivial, as the representation should be language independent and as these expressions must be type checked. In the current implementation, the translator skips `condition` elements. Therefore, in order to execute translated code containing `condition` elements, the programmer must manually program the corresponding boolean expressions.

Services' operations also require manual modifications of the translated code. Indeed, the reader noticed that SMoL models services in terms of peer's interaction (i.e. `InformationHandler`s, `receiveMessage()`, `invoke()`, etc.). However, being a specification language, SMoL does not feature any mechanism to actually model the body of service operations. For instance, it is possible to model the way a thermometer publishes the temperature it reads but it is not possible to model the actual reading using SMoL. Therefore, once the translated SMoL code is achieved, the programmer must manually modify it to include the corresponding code. Note that these modifications are inherent to the nature of SMoL and are not the consequence of its XML representation.

In conclusion, this gap is the reason why we have to scope the features of the translator. It is not possible, because of the nature of SMoL and because of the issues induced by its XML representation, to produce a fully executable translation from a SMoL specification. However, we believe that the additional code modifications are, on the one hand, necessary because of SMoL's nature and, on the other hand, negligible in terms of programming efforts.

[2]This is required as the data are represented by the type `tFrom`. Therefore, to identify a literal value, it must be encapsulated within a uniquely identifiable data type.

4.1.3 Link with the API

Because of the way exceptions are handled in SMoL and the synchronisation requirements of some commands, the middleware API needs information from the translator. In particular, the primitives must be able to check whether a fault has been raised or not to terminate their execution if it is the case. In addition, the primitives invoked inside a `Pick` or an `InformationHandler` need to be aware of the status of their parent command. This is modelled by providing the reference of the parent command as parameter for each primitive. By using this reference, it is possible to call methods exhibited by the `ThreadedCommand` interface since all command objects implement it.

4.2 SMEPP middleware API

In order to present how our implementation of the primitives has been made, we firstly introduce how the SMEPP basic concepts are modelled using SecureLime ones. Then, we will outline how the software has been designed to comply with the SMEPP requirements. Finally, we detail the implementation of the primitives.

4.2.1 High level design

An important part of SMEPP is the discovery of services and groups. In our implementation, this is done by using two tuple spaces, playing the role of service (*SD*) and group (*GD*) directories. The tuples contained in these two tuple spaces constitute respectively a list of service descriptions and a list of group descriptions. Consequently, the search for a group or a service is achieved by means of a *read* operation on the corresponding tuple space.

A peer is mapped to a Lime Agent[3] which has two default tuple spaces[4]: the *SD* and the *GD* tuple spaces described above.

When a peer wants to create or join a group G, it has to :

- create a group tuple space G,
- put a tuple (called reference-tuple) describing G in *GD*, and
- put a tuple (called membership-tuple) in G to update the group membership list.

A group is a set of peers which have executed these three actions. The members of a same group G use the group tuple space to communicate, as it will be illustrated in the scenarios below. Remember that tuple spaces having the same name (and password) are merged to form a federated one.

A service can be discovered through its service-tuple[5] (providing its contract and identifiers), see Figure 4.9. As for the service implementation, a service is simply a Java-based process using our API.

Basically, an event is modelled by the release of a tuple in a group tuple space *G*. In order to receive an event (`receiveEvent()` primitive), a peer creates a new SecureLime reaction waiting for a tuple corresponding to the right event-tuple, see Figure 4.15.

Figure 4.4 illustrates the modelling of a peer and the groups to which it belongs. One can see the two discovery tuple spaces, *GD* and *SD*. The peer is member of n groups. There is a tuple in *GD* and a tuple space for each one. Each group tuple space contains a membership-tuple representing the peer. Furthermore, the peer exhibits m services, which are listed in *SD*. The figure shows also an example of event-tuple in the group_*2* tuple space.

[3] See Section 3.5.
[4] Created by the execution of `newPeer()`.
[5] Which is inserted in *SD* after a `publish()` call.

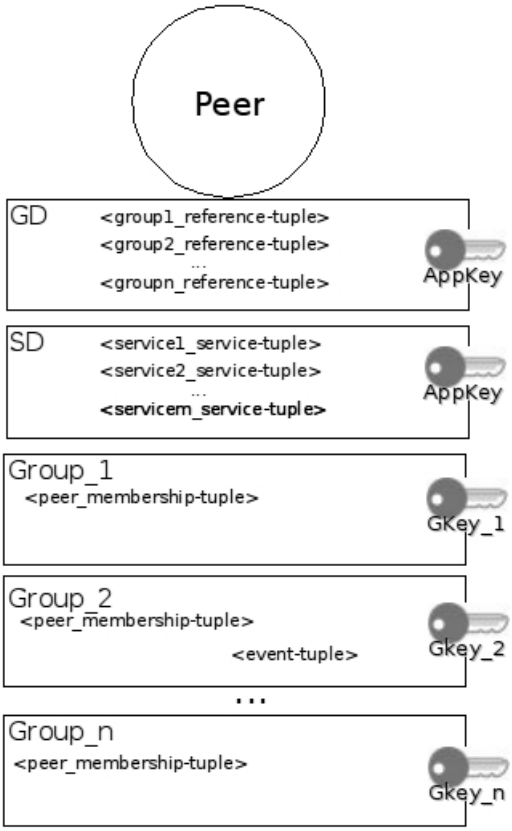


Figure 4.4: Peer model.

4.2.2 High level security design

The security aspects of our implementation have been addressed by using the SecureLime's extensions of Lime. According to the SMEPP guidelines:

- every peer has an AppKey password granting access to a SMEPP application,
- every peer has a set of passwords (GKeys) granting access to groups.

In order to prevent illegal peers to get access to the SMEPP application, the directory tuple spaces (*SD* and *GD*) are protected by using SecureLime secured tuple spaces with the AppKey as password. In this way, every data passing through these tuple spaces is encrypted with the AppKey.

Note that if the peer does not provide the right password at peer creation (`newPeer()` primitive), it does not get an error message. Actually, it creates isolated directories (since the secured tuple spaces do not merge if they do not have the same password, thanks to the SecureLime federation mechanism). Furthermore, if an illegal peer creates a group, legal peers will not be able to see it since they will not share the same directories.

To ensure that every peer sees only groups and services matching its credentials, we had to prevent the tuples inside the directories from being visible to everyone. SecureLime made this task pretty easy. It suffices to use the password of the group as read-password, thus a peer is only able to see a group or service description if it has the password matching the group or service visibility.

To restrict the access to a group, we protect the secured tuple space representing the group with the GKey corresponding to it. In this way, when a peer tries to join a group, if it provides the right password, the newly created tuple space is merged with the federated one. Otherwise, it gets an empty tuple space, as in the case of a `newPeer()` call with an incorrect password.

Figure 4.4 shows the keys used to protect a SMEPP application. The discovery tuple spaces are protected with AppKey, while each group tuple spaces is protected with its own associated GKey, GKey_i in the figure.

4.2.3 Software architecture

Figure 4.5 shows the architecture of our software. This section explains more in details the role of each package. Please note only the principal classes are represented in this class diagram.

Utilities package

In the `utils` package, one can find the `TupleFactory` which is a class gathering every method producing a tuple. This enables the programmer to change the tuple structure in a consistent way for each class of tuple. The `TupleFactory` is thus used by many classes.

The `ids` subpackage contains the identifiers used in the API. Those objects refer to the identifiers defined in the SMEPP Service Model [98], for instance `GroupServiceID` corresponds to `groupServiceId`. Let's have a closer look to their structure.

- `PeerID`. This object has only one attribute: the `AgentID` returned by the SecureLime API.
- `GroupID`. Since several different groups can have the same name in SMEPP, we have to use more information to have a proper identifier. A `GroupID` is then composed of the name of the group, the creator's `PeerID` and a counter (local to the creator). This last attribute is incremented every time the peer creates a group.
- `ServiceID` is an interface implemented by `PeerID`, `SessionID`, `PeerServiceID` and `GroupServiceID`. This interface finds its utility in primitives such as `invoke()`, `getPeerId()` etc. It simply enables to use the same primitive implementation for different classes of argument.
- `SessionID` is an object containing several components: the `PeerID` of the session initiator, the `PeerID` of the session-full service provider, the service's `PeerServiceID` and a counter (to ensure the object to identify uniquely the session).
- `Caller` is an interface implemented by `PeerID`, `SessionID` and `PeerServiceID`. Those entities run a piece of SMoL code. Thus, they can invoke operations. The interface in itself is used to identify the sender of an event.
- `CallerID` has three attributes: an `AgentID`, a `GroupID` and an object which implements `Caller`. When an entity invokes an operation, it uses this object to enable the provider to reply in the right (local) tuple space.
- `GroupServiceID` consists of a string composed of the `GroupID` and a hashcode of the service contract. This ensures that every service having the same contract and published in the same group gets the same `groupServiceId`.
- `PeerServiceID` is an object with two attributes: the corresponding `GroupServiceID` and the `PeerID` of the peer offering the service.

Peer package

The root of the `peer` package contains:

- the `Peer` class which implements SecureLime's `StationaryAgent` interface. It is used to contain the data hold by a SMEPP peer. Notably, it holds a boolean variable to ensure the peer is connected to the SMEPP application, the peer identifier (`PeerID`), etc.,
- the `Primitives` class which gathers the implementation of all the primitives,

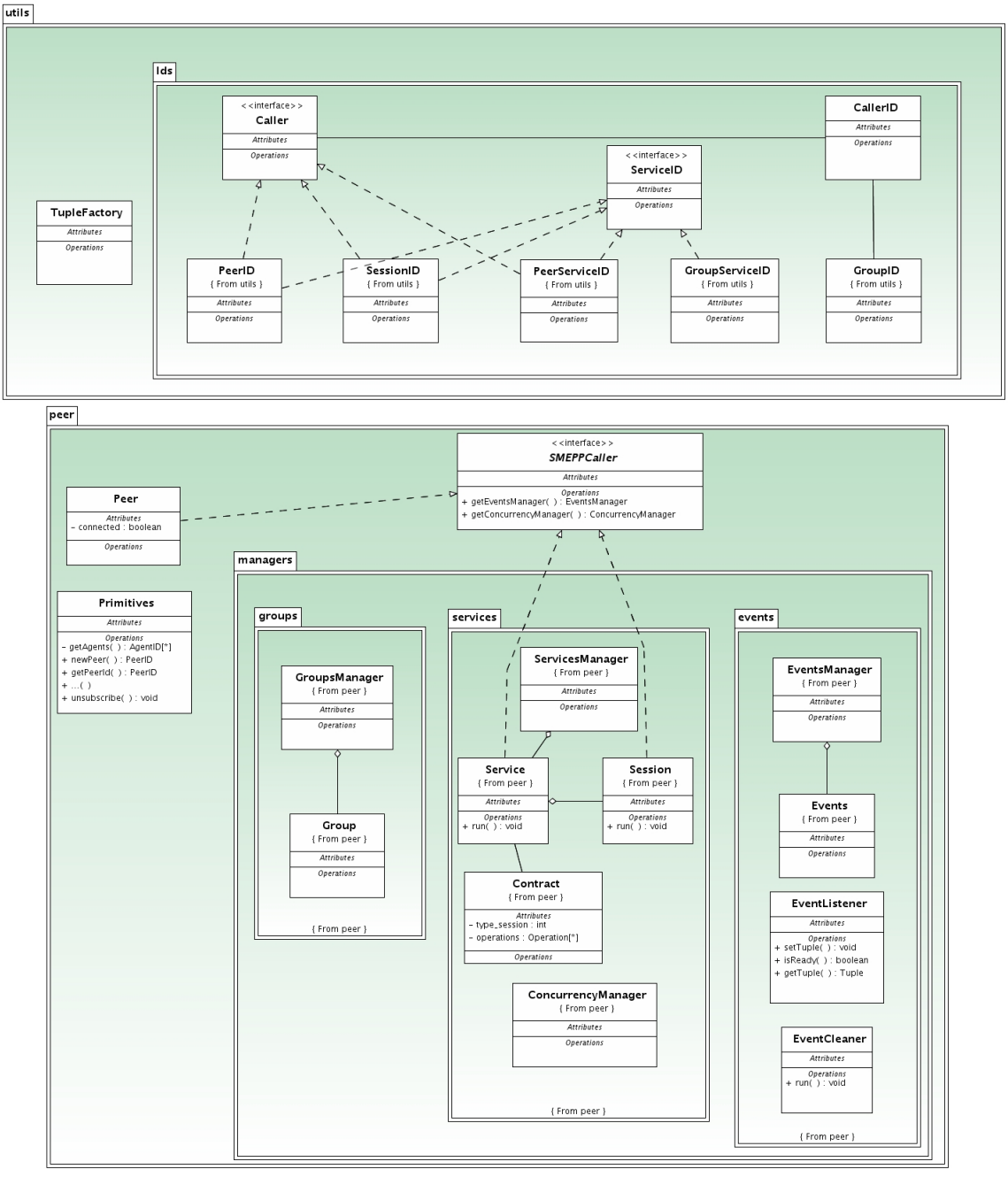


Figure 4.5: Primitives API.

- the `SMEPPCaller` interface which is implemented by the "running" classes (i.e. `Peer`, `Service` and `Session`). It provides two methods which return the event and concurrency manager (respectively) associated with the entity.

The package contains also a subpackage (`managers`) which gathers the managers of a peer. Those classes are (notably) managing the membership of groups, the services offer and the events subscriptions. We describe them in details in the following.

Groups. The role of this part of the program is to manage the groups to which the peer belongs. When a peer creates or joins a group, the group is simply added to the list. In this way, the peer can access the group tuple space easily. Also, when a peer leaves a group, the group manager is in charge of disengaging the tuple space and cleaning all other informations related to this group.

Services. The service management package manages the service publishing, the session creation, etc. This is a very important part of the API. Indeed, a service is also a SMEPP entity and can thus call primitives to receive events, invoke operations, etc. If the service is session-full it can have sessions which are the running instances of the service. Both `Service` and `Session` extend the `LimeThread` class since SecureLime restricts access to tuple spaces to this class. Note that in case the service is session-full, the running sessions are represented as objects contained in its `Service` object.

This package contains also the `ConcurrencyManager` which is in charge of preventing

- a same entity to call the same method twice at the same time by keeping a list of `<entityId, operationName>` couples,
- an entity to call `reply()` without having invoked a corresponding `receiveMessage()`. The class keeps a list of `<callerId, operationName>` to keep track of received message.

Events. This part of the API manages the event subscriptions. In SMEPP, one can subscribe to all events at the same time and unsubscribe to them event by event, and symmetrically. The implementation of such a mechanism is quite complicated since there is no way of knowing all available events. The `EventsManager` keeps a list of `Event`. This object represents a (un)subscription to one event in a group, one event in any group, all events in a group or all events in any group. For instance, if a peer subscribes to all events, we add a special *event* "all". Then, if the peer unsubscribes to "eventName", we add an *anti-event* "eventName".

This subpackage contains also two important classes. The `EventListener` class (which implements `ReactionListener` from SecureLime) is used to retrieve an event with the `receiveEvent()` primitive. The `EventCleaner` is a thread created after the release of every event, it deletes the event after a few seconds to ensure the freshness of every type of event.

4.2.4 Primitives implementation

This section details how the SMEPP primitives are implemented with Java on top of SecureLime. Each primitive is described from four points of view (when relevant). Firstly, they are described as SMEPP defines them. In this way, the reader can fully understand what is needed to implement them. Secondly, the coordination point of view of the implementation is outlined, referencing SecureLime. Thirdly, some implementation details are given. Finally, the primitives' exception management is described.

The primitive are organised as in Section 2.3.1. We start by presenting the peer management related primitives. Then, we detail the ones concerning the group management. Subsequently, we dwell on the service related primitives. The two last parts details the message and event related primitives. Some scenarios allow one to understand completely the interaction underlying the main primitives. In the following, three important scenarios illustrate the implementation, regarding group, service and event management.

Peer management

The three peer management primitives are `newPeer()`, `getPeerId()` and `getPeers()`.

NewPeer

```
peerId newPeer(credentials) throws exception invalidCall
```

Primitive description. A program calls the `newPeer()` primitive to become a SMEPP peer. The `credentials` parameter is used to authenticate the peer. In our implementation it consists of the so-called *AppKey* and a list of pairs `<GKeys,groupName>`. A call to `newPeer()` returns a `peerId` which identifies the peer in the SMEPP application. The *invalidCall* exception is thrown when a service tries to call the primitive.

Coordination view. As outlined above, the implementation of `newPeer()` consists mainly of the creation of two tuple spaces: *GD* and *SD*, which represent the group directory and the service directory, respectively. Those tuple spaces are created using standard names and the provided *AppKey*. The `peerId` is produced by using the `AgentID` given by SecureLime which is unique for each host running a SecureLime server.

Implementation details. Inside a peer code, no primitive can be called before `newPeer()`. A boolean variable (set to `true` at the end of the method) is used to check that the peer is well connected to the SMEPP application when invoking other primitives.

Exceptions handling. At the beginning of the method implementing the primitive, the API checks which kind of "container" is the caller. Indeed, if the "container" is a service or a session, the call is invalid and an *invalidCall* exception must be raised. This is done by using the `getContainer()` method defined in the `SMEPPCaller` interface. This check is done in other primitives available only to peers.

GetPeerId

```
peerId getPeerId(id?)  throws exception invalidId
```

where `id` is either a `peerServiceId` or `sessionId`

Primitive description. This primitive behaves differently depending whether the `id` parameter is provided or not. If an `id` is specified, then it returns the `peerId` of the peer which has published the service referred by the `peerServiceId` or the `sessionId`. When the primitive is called without parameter, the primitive returns simply the `peerId` of the peer containing the caller. An exception is raised if the `id` does not refer to an existing service or session.

Implementation details. SecureLime is not required to implement the core of this primitive since the object representing a `peerServiceId` in our implementation contains the provider's `peerId`. Similarly, the `sessionId` contains the `peerServiceId` of the instantiated service.

Exceptions handling. In order to be able to raise the *invalidId* exception, we have to scan the *SD* to find a service-tuple (see Figure 4.9), by using a `rd()` operation. If the tuple cannot be found, this means the corresponding service or session does not exist, the exception is then raised.

GetPeers

```
peerId[] getPeers(groupId) throws exception invalidGroupId, callerNotInGroup
```

Primitive description. The `getPeers()` primitive returns the list of peers which are member of the group `groupId`. The *invalidGroupId* exception is raised if no group is identified by `groupId`. *CallerNotInGroup* is raised if the group exists but the caller peer is not a member, or if the caller service is not published in this group.

<'_peer',peerId,#,peerPwd>		
where	peerId	is the identifier of the tuple owner,
	#	means no read-password is used,
	peerPwd	is the remove-password generated by the peer.

Figure 4.6: Membership-tuple.

Coordination view. The core implementation is trivial. Since every group member has put a membership-tuple (see Figure 4.6) in the group tuple space, a simple `rdg()` operation[6] retrieves an array of tuples. This array can be used (almost) directly to return the expected result.

SecureLime restriction. SecureLime made things slightly harder by requiring to specify the location parameter in the `rdg()` operation. Since every membership-tuple is in a different location (remember those tuples are located in the local tuple space of each peer), we have to make a `rdg()` operation on each connected agent (i.e. peer) specifying the location parameters.

In order to get the list of locations in which one has to search, one uses the `LimeSystemTupleSpace`. Remember this tuple space contains, among other things, a tuple for each connected agent. From this tuple, we can extract the agent's `AgentID` and, then, compute the associated location.

This request has been encapsulated in a method called `getAgents()` and is used in other primitives, every time an aggregate or probe operation on a whole tuple space is needed.

Exceptions handling. To comply with the exceptions specified in the primitive signature, two checks are needed. Firstly, we have to query the *GD* tuple space to ensure that the `groupId` group exists. This is done by using the `getGroupDescription()` primitive. Secondly, the API has to ensure that the peer is a member of the `groupId` group or the service has been published in this group. This last check is done "internally" in the API by using the `GroupsManager` and `ServicesManager`.

Group management

There are seven primitives which take the group management in charge: `createGroup()`, `getGroups()`, `getGroupDescription()`, `joinGroup()`, `leaveGroup()`, `getIncludingGroups()` and `getPublishingGroup()`. The implementations of these primitives make above all use of tuple space creations and reference-tuples.

CreateGroup

`groupId createGroup(groupDescription) throws exception invalidCall`

where `groupDescription` stands for `<groupName,securityInformation>`

Primitive description. A peer calls `createGroup()` to create a new SMEPP group, this group will be accessible for every peer having the corresponding password. The `groupDescription` parameter encapsulates the group name, some security information and, possibly, some further information (e.g. a textual description). However, in our implementation the `securityInformation` parameter is not used since the access to a group is simply restricted by the ownership of a password. Thus, to be able to access and see a group G, one only needs the password associated with G. One can say that the security information is inferred from the group name. Practically, the `groupDescription` parameter consists of two parts, a (mandatory) name and an (optional) textual description. A service is not allowed to call the `createGroup()` primitive, consequently such a call raises an *invalidCall* exception.

[6] This operation uses a template corresponding to a membership-tuple.

<'_group',name,groupId,groupDescription,peerId,groupPwd,peerPwd>		
where	name	is the group name,
	groupId	is the group identifier,
	peerId	is the id of the tuple owner,
	groupDescription	is the textual description of the group,
	grouPwd	is the read-password corresponding to name,
	peerPwd	is the remove-password generated by the peer.

Figure 4.7: Reference-tuple.

Coordination view. Basically, the translation of this primitive into SecureLime concepts is simply the creation of a new tuple space using the wished `groupId` and the corresponding password as parameters. The `groupId` is computed using the `peerId` of the creator, the group name and a counter associated with the peer.

In addition, the peer adds a membership-tuple (see Figure 4.6) into the newly created tuple space. This tuple is used to manage the membership list of the group. Then, the peer has to add a reference-tuple (detailed in Figure 4.7) in the *GD* tuple space in order to make the group discoverable. Those actions are implemented using `out()` operations.

Exceptions handling. As usual, the *invalidCall* exception is raised if the "container" is not a SMEPP peer (by using the `getContainer()` method).

GetGroups

```
groupId[] getGroups(groupDescription?)
```

Primitive description. This primitive returns an array of group identifiers. The `groupDescription` parameter is used to filter the result. The SMEPP service model allows to filer on the group name, the description and the security information. However, as stated before, this last parameter is not used in our implementation. Instead, the primitive searches for every group matching the name and/or description using every password the peer owns.

Coordination view. Since every group is referenced in the *GD* tuple space[7], the peer has to query it to retrieve group information. This query is implemented by successive `rdg()` operations on the tuple space. The number of `rdg()` calls is equal to the number of *GKey*s the caller owns. The template used in `rdg()` corresponds to the optional `groupDescription` parameter. If no parameter is specified, the operation returns all tuples the peer has access to.

Once all pertinent reference-tuples are retrieved, it suffices to extract their `groupId` field and to return them as an array of `groupId`s. See Figure 4.7 for more details on reference-tuples.

SecureLime restriction. In this case again, we have to face the SecureLime restriction on the aggregate operations. So, as in `getPeers()`, we need to know the list of the connected agents first and then iterate on the agent list specifying their location in the `rdg()` operation.

Implementation details. Since every peer has a reference-tuple (in its local *GD*) for each group of which it is member, the `rdg()` operation potentially returns several times the same `groupId`. To avoid this situation, a selection is made before returning the `groupId` array to erase duplications.

GetGroupDescription

```
groupDescription getGroupDescription(groupId) throws exception invalidGroupId
```

[7] In the form of reference-tuples.

Primitive description. `GetGroupDescription()` returns the description of a group. In our implementation, it consists of the name and (potentially) the textual description of `groupId`. It raises an *invalidGroupId* exception if no corresponding group can be found.

Coordination view. The implementation of the primitive is straightforward. Indeed, to retrieve a group description it suffices to retrieve the reference-tuple corresponding to that group. We use a `rdp()` operation with a template specifying the `groupId` to find a corresponding tuple. As usual, since those tuple are read-protected by the group password, we have to iterate on the peer's password list to return the `groupDescritpion` accordingly to all the `GKeys` owned by the caller.

Exceptions handling. If no matching reference-tuple can be found, an *invalidGroupId* exception is raised.

JoinGroup

`void joinGroup(groupId, credentials) throws exception accessDenied, invalidGroupId, invalidCall`

Primitive description. A peer uses `joinGroup()` to enter a SMEPP group. It has to provide the `groupId` and its `credentials` (i.e. in our implementation, the list of <`group name, password`> pairs). Remember that a peer can join several groups. Moreover, joining several time the same group does not raise an exception.

The service model specifies three throwable exceptions: *accessDenied* if the peer does not have the right credential, *invalidGroupId* if the group does not exist and *invalidCall* if the caller is a service.

Coordination view. The implementation of the primitive is very similar to the `createGroup()` one. Indeed, the peer creates a tuple space using the `groupId` and the password corresponding to the group name. Then, it puts a reference-tuple (Figure 4.7) in *GD* and a membership-tuple (Figure 4.6) in the newly created tuple space.

To fill those two last tuples, the `groupId` is not enough. That is why, the primitive firstly fetches the corresponding reference-tuple and extracts the name and the (potentially empty) description.

Exceptions handling. On the one hand, the *invalidGroupId* is raised if the group description cannot be found in *GD*. On the other hand, the *invalidCall* is raised, as usual, if the "container" is not a peer. Regarding the *accessDenied* exception, it is important to signal that if the peer has a wrong password associated with the name of the group, it will not get an exception. There is no way of preventing a peer to create a new protected tuple space, but remember this tuple space will merge with another one only if they have the same name *and* the same password. In our case, if the peer uses a wrong password, it will create a "new" but *isolated* SMEPP group. The *accessDenied* exception is then raised if no password is associated with the group name[8] in the `credentials`.

LeaveGroup

`void leaveGroup(groupId) throws exception invalidGroupId, peerNotInGroup, invalidCall`

Primitive description. A peer uses the `leaveGroup()` primitive to exit a group it belongs to. This primitive is also in charge of removing all services a peer has offered through the group it wants to leave. Note that "if a peer leaves a group while having active service instances in the respective group, then the middleware will not raise an exception. In such a case service invokers will (possibly) receive an exception" [98].

As a service is not allowed to call the `joinGroup()` primitive, it is not allowed to call `leaveGroup()`. This raises an *invalidCall* exception. Furthermore, if the `groupId` group

[8] Found in the reference-tuple.

does not exist, it raises an *invalidGroupId* exception. If the peer is not a member of the group, a *peerNotInGroup* exception is raised.

Coordination view. In order to exit a group, a peer has to *disengage* the group tuple space corresponding to `groupId`. This deletes all the tuples concerning events and services in this group. It has also to delete the `groupId` reference-tuple from *GD* and the service-tuples representing the services it has published in this group from *SD*. This is done by using `inp()` operations.

Implementation details. To stop the execution of the services published in the `groupId` tuple space, the peer has to stop the execution of the root `FaultHandler` of every concerned service (and possibly session). It uses the `stopCmd()` method[9] to terminate the execution of the `FaultHandler`'s body.

Exceptions handling. The *invalidCall* is, as usual, raised if the "container" is not a peer. A call to `getGroupDescription()` is made to ensure the group exists, otherwise the *invalidGroupId* exception is raised. The last check ensures the peer is well a member of the group using the `GroupsManager`, otherwise one raises the *peerNotInGroup* exception.

GetIncludingGroups

```
groupId[] getIncludingGroups() throws exception invalidCall
```

Primitive description. `GetIncludingGroups()` returns an array containing the groups' identifiers to which the caller peer belongs. A service is not allowed to call this primitive. Such a call raises *invalidCall*.

Implementation details. The implementation of this primitive is Java-based only since the peer itself has all the information it needs. It returns the array thanks to the information held by the `GroupsManager`.

Exceptions handling. The *invalidCall* primitive is raised if the caller "container" is not a peer.

GetPublishingGroup

```
groupId getPublishingGroup(id?)  throws exception invalidId, invalidCall
```

where `id` is either `groupServiceId`, `peerServiceId` or `sessionId`.

Primitive description. The primitive behaves differently according to whether the parameter is specified or not. If `id` is not provided, the primitive returns the group in which the caller service has been published. Note that this implies a peer cannot call the primitive without parameter (in such case, one raises an *invalidCall* exception). If the `id` parameter is specified, the primitive returns the `groupId` in which the service identified by `id` has been published.

Implementation details. The implementation of the primitive is really simple since our API defines the `groupId` as a part of a service identifier: `peerServiceId`, `groupServiceId` and `sessionId` have a `groupId` component.

Exceptions handling. Actually, this primitive uses SecureLime if the `id` parameter is specified. Indeed, in this case, one has to check the service identified by `id` really exists. This is done by *trying* to retrieve a service-tuple matching `id` from *SD* with a `rdp()` operation.

[9] Declared in `ThreadedCommand` interface.

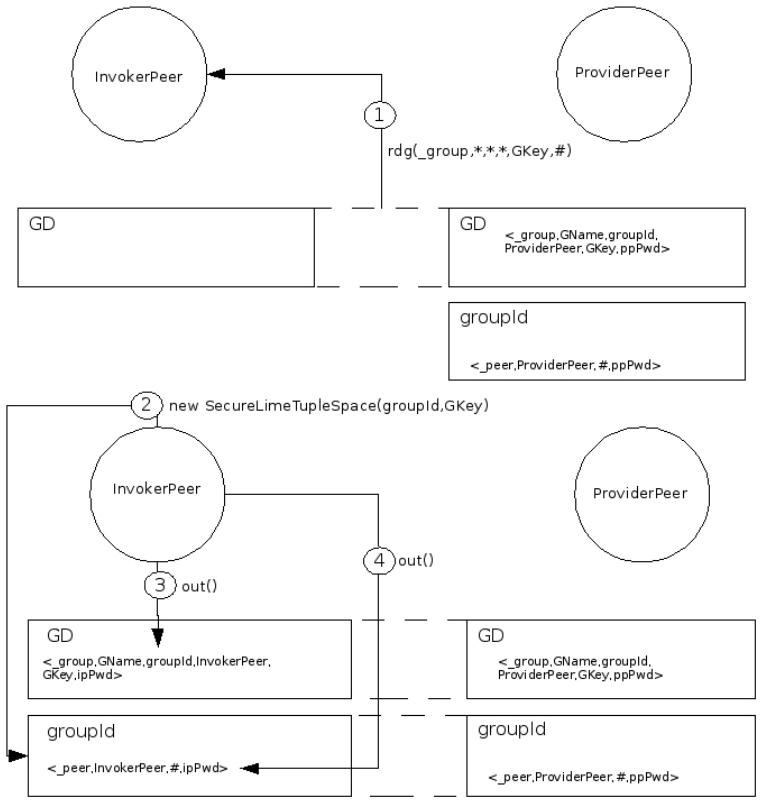


Figure 4.8: Group management.

Group management scenario

We assume that the *ProviderPeer* has created a group called `GName`, identified by `groupId`. The *InvokerPeer* will execute "`gid[] = getGroups()`" and "`joingroup(gid[0],myCredentials)`" to join this group. We also assume that the peers share the AppKey and the password to get access to *groupId*, *GKey*. In Figure 4.8, only one group (*groupId*) has been created (by *ProviderPeer*), a reference-tuple references it in *GD*. This tuple contains the name of the group, its identifier and the reference owner. It is read-protected by *GKey* and remove-protected by *ppPwd*[10]. There is also a membership-tuple for *ProviderPeer* in *tempGroupId*. This one contains the peer's identifier.

1. *InvokerPeer* invokes `getGroups(desiredGroupDescription)` to discover which groups it can join. To do this, it performs a read operation on the federated *GD* tuple space using *GKey* as read-password. With regards to the `rdg()` operation, `desiredGroupDescription` corresponds to the name of the group (not used here). *InvokerPeer* gets the only tuple present which contains the group name and its identifier which are the needed information to join the group later.

2. *InvokerPeer* invokes `joinGroup(groupId[0],myCredentials)` which creates the new `SecureLimeTupleSpace` object, representing the *groupId* group tuple space. The peer has to use the group identifier (*groupId[0]*) and the password corresponding to it (*GKey*). Since the two *groupId* tuple spaces have the same name and password, they automatically merge to create a federated tuple space.

3. `joinGroup()` continues by putting a reference-tuple describing the group in the *GD* tuple space. This step ensures that the group will be kept alive as long as the group contains at

[10]Remember that the last two fields of every tuple are respectively the read and remove-password. Here, *ppPwd* (which has been securely generated by the peer) is used to ensure that only the tuple's owner can remove it.

<'_service',contract,gsid,psid,groupId,peerId,groupPwd,peerPwd>		
where	contract	is the service contract (in a Java object form),
	gsid	is the service groupServiceId,
	psid	is the service peerServiceId,
	groupId	is the group identifier of the group in which the service has been published,
	peerId	is the id of the service provider,
	grouPwd	is the read-password corresponding to the service's group,
	peerPwd	is the remove-password generated by the peer.

Figure 4.9: Service-tuple.

least one member. This tuple differs from the *ProviderPeer*'s one by the remove-password field (*ipPwd*) and the field expressing the tuple owner.

4. The last step of `joinGroup()` consists of putting the peer "membership-tuple" into the *groupId* tuple space. The purpose of this tuple is to keep updated the group members list.

Service management

Five primitives form the service management: `publish()`, `unpublish()`, `getServices()`, `getServiceContract()` and `startSession()`. The creation of services and sessions involves Java reflection [100] and the publication of service-tuples, while the service discovery is implemented by performing service-tuple retrievals.

Publish

```
<groupServiceId, peerServiceId> publish(grouId, serviceContract) throws exception
invalidService, invalidGroupId, peerNotInGroup, invalidCall
```

Primitive description. This primitive is used by peers which want to offer a service (defined in the `serviceContract`) in a group, `groupId`. It returns two identifiers: the `peerServiceId` which identifies the instance of the service offered by the peer and the `groupServiceId` which allows a SMEPP entity to invoke a service "blindly". Remember that the services having the same `serviceContract` and published in the same group have the same `groupServiceId`.

`Publish()` raises four kinds of exception:

- *invalidService* if the service contract does not refer to a valid service,
- *invalidGroupId* if `groupId` does not refer to an existing group,
- *peerNotInGroup* if the caller peer is not a member of `groupId`,
- *invalidCall* if the primitive is called by a service.

Coordination view. From the coordination point of view, publishing a service consists of adding a service-tuple in the *SD* tuple space. Service-tuples are detailed in Figure 4.9. This tuple makes the service discoverable for peers having the right password (the same as the group in which the service has been published). Thus, using an `out()` operation, a service-tuple is added in the peer's local *SD* tuple space.

Implementation details. Actually, the implementation of this primitive is slightly more complicated. Indeed, before making the service discoverable, it is important to start its execution.

The way a service is executed depends on its property of being session-full or not. On the one hand, if the service is session-full, the body of the service is started after an invoker calls a corresponding `startSession()`. In this way, the execution of a session-full service amounts to wait for a session-tuple. The detailed mechanism of a session creation will be

explained along with the `startSession()` primitive. On the other hand, if the service is not session-full, its body can be directly executed. We use the Java reflection [100] to load the class representing the service. Remember that this class is actually a `FaultHandler`. Once this class is loaded, the service's thread is started.

Exceptions handling. The following lines explain how the exceptions are handled in the implementation of this primitive.

- *invalidService* is never raised by our API since the service contract is checked at the "translation time",
- *invalidGroupId* is raised if the group description of `groupId` cannot be found in *GD*,
- *peerNotInGroup* is raised if `groupId` is not in the group list to which the peer belongs,
- *invalidCall* is raised when `getContainer()` returns that the caller is not a peer.

Unpublish

`void unpublish(peerServiceId) throws exception invalidServiceId, peerNotServiceOwner, invalidCall`

Primitive description. To stop offering a service, a peer calls the `unpublish()` primitive, specifying the `peerServiceId` corresponding to the service it wants to stop.

If the peer is not the owner of the service it wants to unpublish, one raises a *peerNotServiceOwner* exception. In case the `peerServiceId` does not refer to an existing service, one raises an *invalidServiceId* exception. Finally, similarly to the `publish()` case, one raises an *invalidCall* exception if the primitive is called by a service.

Coordination view. The unpublishing of a service is done by deleting the corresponding service-tuple from the *SD* tuple space, using an `inp()` operation.

Implementation details. As with `leaveGroup()` the service execution has to be stopped, as well as the potentially running sessions. The execution is stopped by invoking the `stopCmd()` method on the root `FaultHandler` representing the service.

Exceptions handling. Checking if the peer is the owner of the service is trivial. However, to ensure to raise the *invalidServiceId* when needed, one has to query the *SD* tuple space to find a corresponding service-tuple using a `rdp()` operation. As usual, *invalidCall* is raised if the "container" is not a peer.

GetServices

`<groupId, groupServiceId, peerServiceId>[] getServices(groupId?, peerId?, serviceContract?, maxResult?, credentials) throws exception invalidGroupId, invalidPeerId, invalidService`

Primitive description. The goal of this primitive is to retrieve the service identifiers matching the specified parameters (all optional excepting `credentials`). It returns an array of triples (with max `maxResult` elements) consisting of `groupId`s, `groupServiceId`s and `peerServiceId`s. Note that in our implementation the matching of the contract is only based on the session type and the method names.

Coordination view. The implementation of the primitive is very similar to the `getGroups()` one. Indeed, a `rdg()` operation using a template corresponding to the specified parameters is invoked on *SD* to retrieve the matching service-tuple (Figure 4.9). As the services have different read-protected passwords, one has to iterate on the password list contained in the `credentials`.

SecureLime restriction. As in `getGroups()`, one has to specify the location parameters in the `rdg()` operation. That is why, the primitive iterates on all the connected agents to retrieve the matching tuples.

<'_session',peerServiceId,sessionId,#,#>		
where	peerServiceId	is the peerServiceId of the aimed service,
	sessionId	is the sessionId computed by the caller,
	#	means no read-password is used,
	#	means no remove-password is used.

Figure 4.10: Session-tuple.

Exceptions handling. As in the `publish()` case, our implementation does not raise the *invalidService* exception. However, *invalidGroupId* is raised if no `groupId` reference-tuple can be found.

GetServiceContract

`serviceContract getServiceContract(id) throws exception invalidId`

where `id` is either `groupServiceId`, `peerServiceId` or `sessionId` .

Primitive description. This primitive returns the contract corresponding to the specified `id`. If the parameter does not correspond to an existing service, the *invalidId* exception is raised.

Coordination view. The implementation is straightforward. A `rdp()` operation on the *SD* tuple space retrieves a matching tuple. Then, one returns the contract contained in the tuple. Though, as the visibility of services is restricted by passwords, one has to iterate on the list of passwords the peer owns.

SecureLime restriction. As in some of the previous primitives, we also have to iterate on the connected agents since the location parameter is required with the "probe" operations.

Exceptions handling. If no matching tuple can be found, one raises the *invalidId* exception.

StartSession

`sessionId startSession(serviceId) throws exception invalidServiceId, accessDenied, cannotStartSession`

where `entityId` is either `groupServiceId`, `peerServiceId` of a session-full service .

Primitive description. A SMEPP entity calls the `startSession()` primitive to open a new session with a session-full service. The primitive returns a `sessionId` which identifies the session uniquely and allows it to be shared amongst several entities.

Three kinds of exception can be raised:

- *invalidServiceId* if no state-full services are referenced by `serviceId`,
- *accessDenied* if the caller entity does not belong to the group in which the service is published,
- *cannotStartSession* when the service cannot start a new session.

Coordination view. In order to request the creation of a new session, the caller entity sends a session-tuple in the local group tuple space of the provider, using an `out()` operation with the location parameter specified. Then, the caller waits for a session-ack-tuple. Those two tuples are detailed in Figures 4.10 and 4.11, respectively. Assuming the provider has published a corresponding session-full service, it created a `LimeThread` waiting for session-tuples. Once the provider has got the needed information, it answers to the caller using a session-ack-tuple acknowledging the creation of the session. This tuple is sent in the caller's local group tuple space. When the caller gets the expected session-ack-tuple, it knows it can now use the session.

		<'_sess_ack',psid,sessionId,bool,#,#>
where	psid	is the peerServiceId of the concerned service,
	sessionId	is the sessionId computed by the session originator,
	bool	is the provider answer (if false, the session can not be started),
	#	means no read-password is used,
	#	means no remove-password is used.

Figure 4.11: Sessionack-tuple.

Implementation details. For the sake of simplicity, it is the caller which computes the `sessionId`. It consists in the identifiers of the caller, the provider and the `peerServiceId`. The provider checks the `sessionId` validity when it receives the session-tuple.

On the provider side, the session is executed as the execution of a service. Once the thread representing the service gets a session-tuple, the provider uses the Java Reflection to load the root `FaultHandler` representing the service. The thread executing an instance of the service is then started. Also, a reference to the session is added to the `Service` object, in order to be able to stop it later (in case of `unpublish()` or `leaveGroup()`).

Exceptions handling. CannotStartSession is raised if the service provider has considered the `sessionId` as invalid, while *accessDenied* is raised if the peer has not the corresponding group in its group list. Finally, *invalidServiceId* is raised either if `serviceId` is not a `peerServiceId` or `groupServiceId`, or if no corresponding service-tuple can be found.

Message management

The `invoke()`, `receiveMessage()` and `reply()` primitives are interconnected with each other. Thus, they use the same format of tuples (invocation-tuples and reply-tuples) to make possible the communication between peers. Following this section, an example scenario helps to understand the mechanism of service invocation.

Invoke

`output? invoke(entityId, operationName, input?) throws exception invalidPeerId, invalidServiceId, invalidOperation, concurrentRequest, invalidInputParameter, invalidOuputParameter, accessDenied`

where `entityId` is either `peerServiceId`, `groupServiceId`, `sessionId` or `peerId`.

Primitive description. The `invoke()` primitive is used to call an `operationName` operation on an entity identified by `entityId`. Remember there are four kinds of invocation depending on the kind of interaction the service offers:

- if the aimed service is state-less, the `entityId` must be either `peerServiceId` or `groupServiceId`.
- if the aimed service is state-full session-less, the `entityId` must be `peerServiceId`, since the `groupServiceId` does not keep track of client.
- if the aimed service is state-full session-full, the `entityId` must be `sessionId`.
- finally, if the caller wants to directly call a peer, the `entityId` must be `peerId`. Note that, in this case, the operation is always assumed *one-way*.

If the type of `entityId` does not comply with the aforementioned invocation restrictions, *invalidServiceId* exception is raised. As one can guess, the `input` and `output` parameters are the input message and output message, respectively. Note that our implementation does not check the type of those elements. Thus, the API never raises *invalidInputParameter* and *invalidOuputParameter* which are supposed to be raised in case the input and/or output do not match the signature of the `operationName` operation.

`Invoke()` behaves differently according to the operation type:

<'_invoke',opName,input,psid,caller,#,#>		
where	opName	is the name of the invoked operation,
	input	is the input operation parameter,
	psid	is the identifier of the aimed service,
	caller	is the identifier of the caller,
	#	means no read-password is used,
	#	means no remove-password is used.

Figure 4.12: Invocation-tuple.

<'_reply',opName,caller,output,fault,#,#>		
where	opName	is the name of the invoked operation,
	caller	is the identifier of the caller (which is waiting for the tuple),
	output	is the output of the operation,
	fault	is the produced fault in case of an operation erroneous behaviour,
	#	means no read-password is used,
	#	means no remove-password is used.

Figure 4.13: Reply-tuple.

- if the operation is *one-way*, `invoke()` blocks until the provider does a corresponding `receiveMessage()`.
- if the operation is *request-response*, `invoke()` blocks until the provider does a `reply()` corresponding to the invoked operation.

The SMEPP model requires that concurrent calls to a same request-response operation by the same entity are forbidden. The primitive raises a *concurrentRequest* exception if the caller entity tries to invoke an operation *Op* while it is concurrently waiting for the response of another *Op* call.

`Invoke()` raises an *invalidOperation* exception if the provider does not support the `operationName` operation and an *accessDenied* exception if the caller does not belong to the provider's group.

Coordination view. The mapping to SecureLime concepts is done as follows. Firstly, the peer inserts (`out()`) an invocation-tuple in the local group tuple space where the service has been published. This tuple is illustrated in Figure 4.12. If the aimed entity is a peer, then the API has to find a common group between the caller and the provider. This is done by iterating on the groups the caller belongs and calling `getPeers()` on them.

Once the peer has sent its invocation-tuple, it waits for a reply-tuple (see Figure 4.13) using an (`in()`). Even in the case of a *one-way* operation, the peer waits for a "fake" reply-tuple. This ensures that `invoke()` blocks until a corresponding `receiveMessage()` is done. In case the operation is *request-response*, this tuple possibly contains the operation's `output`.

Implementation details. In order to ensure that the provider gets enough information to be able to answer the caller, the invocation-tuple contains a `CallerID` Java object which contains the `groupId` used for the communication as well as the caller's `AgentID` and `entityId` (`peerServiceId`, `peerId` or `sessionId`). Without this object, the provider would not be able to answer since it does not know the group in which it has to answer.

SecureLime restriction. An important issue met with SecureLime is the fact that blocking operations (e.g. `in()` and `rd()`) are "passive blocking"[11]. This induces that there is no way to stop a thread which is waiting for such an operation to terminate. However, it is not

[11] "Passive blocking" means the thread does not check regularly whether it can resume its execution.

compliant with the SMoL semantics which requires an execution to stop as soon as an exception is raised.

To get round this problem, we implemented "active waiting" versions of `in()` and `rd()`. Those methods loop every second to make an `inp()` or `rdp()` operation. On each loop, they check if the execution has to be stopped or not (through the `isStopped()` method defined in the `ThreadedCommands`). This allows the API to comply with the specification and also to implement a time-out in case the provider does not answer. Concretely, in this last case a *timeoutException* is raised.

Exceptions handling. To handle the concurrency restriction, the API uses the `ConcurrencyManager` attached to the caller entity. At the beginning of the method, in the request-response case, a pair `<entityId, operationName>` is added in the manager. If such a pair already exists, a *concurrentRequest* exception is raised. When the reply-tuple is received, the pair is removed from the manager, allowing new call of the `operationName` operation.

ReceiveMessage

`<callerId, input?> receiveMessage(groupId?, operationName) throws exception invalidOperation, invalidGroupId, callerNotInGroup, invalidInputParameter`

where `callerId` is either `peerServiceId`, `sessionId` or `peerId`.

Primitive description. Entities use `receiveMessage()` to get an operation request. The primitive returns the `callerId` of the operation caller and the specified `input` message.

The primitive raises four exceptions :

- *invalidOperation* if the operation is not specified in a published service contract (if the caller is a service),
- *invalidGroupId* if no group corresponds to `groupId`
- *callerNotInGroup* if the caller does not belong to the `groupId` group,
- *invalidInputParameter* if the `input` parameter does not match the operation's signature. Note that this last exception is not raised by our implementation.

Coordination view. To receive a message, the caller entity has to make an `in()` operation on the corresponding `groupId` local tuple space. The template of this operation contains the `operationName` and the location parameters specified to look in the appropriate local tuple space. Once the tuple is retrieved, one has to extract the `callerId` and the `input` parameter to return them. In the case `operationName` is *one-way*, the primitive terminates its execution by sending a "fake" reply-tuple. This unblocks the invoker.

SecureLime restriction. Given that the `groupId` is an optional parameter, if it is not specified, the primitive has to explore every group to which the peer belongs. Again, one uses the "active blocking" `in()` operation previously introduced. A loop on groups waits several seconds for a potential invocation-tuple in the current group. The loop continues until a tuple has been found in a local group tuple space.

Implementation details. Since a `reply()` (see in the following) is only allowed if a corresponding `receiveMessage()` has been done before, a pair `<callerId,operatioName>` is added to the caller entity's `ConcurrencyManager`. This allows to check the existence of such a message in the corresponding `reply()`.

Other difficulties arise when the primitive is called through `Pick` and `InformationHandler` SMoL commands. In those cases, the primitive has to be synchronised with other potential `receiveMessage()` or `receiveEvent()` branches. The following details the implementation for those cases, assuming the `groupId` is specified[12].

- **Pick.** While no tuple is found, no other branch is taken and no exception has been raised, the following steps are executed.

[12] Otherwise, the primitive also loops on the groups to which the peer belongs.

1. One does an `rdp()` operation[13] on the group tuple space.
2. If the tuple has been found, one signals it to the parent `Pick` object, through the `setReady()` method[14]. If `rdp()` returns `null`, the thread sleeps for one second and restarts the loop.
3. Then, one executes an `inp()` operation to actually retrieve the tuple located before[15].
4. If the tuple is well retrieved, the parent `Pick` is warned that it can finish its execution through the `setPicked()` method. Otherwise, the loop restarts.
5. The tuple can now be returned.

- **InformationHandler.** The `InformationHandler` case is simpler. One loops while no tuple is found, no exception is raised and the main command is still running (`isFinished()` method). The body of the loop contains only an `inp()` operation and a `sleep()` call in order to retrieve the tuple while avoiding the CPU to be overloaded.

Exceptions handling. *InvalidOperation* is raised if no available (i.e. published) contract contains an `operationName` operation. *InvalidInputParameter* is never raised because of the gap between variable types defined in SMoL and Java objects (see the scope of the thesis in Section 2.4). Other declared exceptions are raised similarly as in previous primitives.

Reply

`void reply(callerId, operationName, output?, faultName?) throws exception invalidPeerId, invalidPeerServiceId, invalidOperation, missingReceiveMessage`

where `callerId` is either `peerServiceId`, `sessionId` or `peerId`.

Primitive description. This primitive terminates the execution of a request-response operation. The caller of `reply()` must have previously invoked a corresponding `receiveMessage()`. Otherwise, a *missingReceiveMessage* exception is raised. The `faultName` parameter is used to signal the `operationName` invoker that an erroneous behaviour has occurred. Exceptions raised by `reply()` are similar to the one raised by `invoke()`.

Coordination view. Basically, `reply()` is implemented by the sending of a reply-tuple to the operation caller's local group tuple space, using an `out()` operation (with location parameters specified). This tuple unblocks the invoker. The reply-tuple format is detailed in Figure 4.13.

Implementation details. We use the `ConcurrencyManager` to ensure that the caller of `reply()` has previously done a corresponding `receiveMessage()`. If a pair `<callerId, operationName>` cannot be found, the primitive is stopped and a *missingReceiveMessage* exception is raised.

Exceptions handling. The exception handling is the same as in `invoke()`.

Message management scenario

The following scenario illustrates the invocation of an operation provided by the *TempReaderService* service. We assume that two peers take part in the SMEPP application. *ProviderPeer* offers a session-full service identified by *psid*. *InvokerPeer* wants to invoke an operation (*opName*) exhibited by the service contract of *psid*. Here are the SMEPP primitives executed by the peers:

InvokerPeer	*ProviderPeer*
`sessionId=startSession(psid);`	`<cid,in>=receiveMessage(groupId,opName);`
`tmp=invoke(sessionId,opName,'input');`	`reply(cid,opName,'output',-);`

[13] Remember the invocation-tuple is sent in the local group tuple space of the provider.
[14] This method blocks until no other branches are "ready".
[15] If the tuple is different, it does not matter.

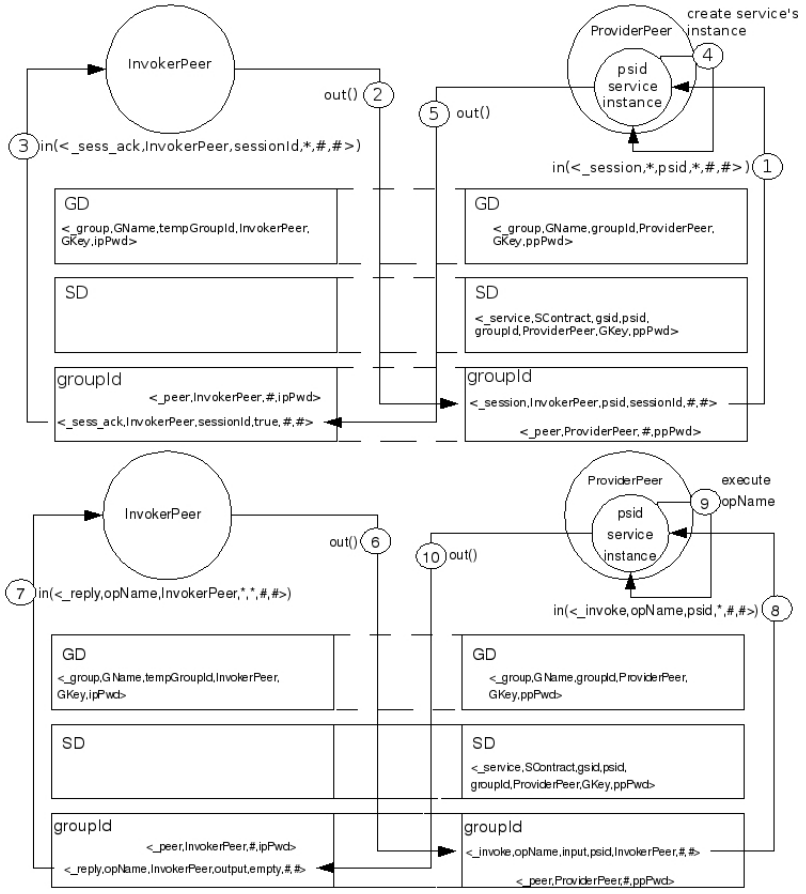


Figure 4.14: Service invocation.

Figure 4.14 shows that both peers are members of the *groupId* group since they have the *groupId* tuple space, the reference-tuple and the membership-tuple. *ProviderPeer* has published the *psid* service[16], thus there is a tuple in *SD* containing the service contract, the peer service identifier (*psid*), the group service identifier (*gsid*), the group in which it has been published and the provider. It is read-protected by *GKey* and remove-protected by *ppPwd*, a password generated by *ProviderPeer*.

1. The first action represents what directly follows the publication of a session-full service. The *ProviderPeer* waits a session-tuple by doing an `in()` operation on its local group tuple space. The template of this tuple contains *psid*, the service identifier.

2. *InvokerPeer* requests a session creation by calling `startSession(psid)`. This puts a session-tuple in the local *groupId* tuple space of *ProviderPeer*.

3. Then *InvokerPeer* waits its session to be acknowledged by the provider. This is done by doing an `in()` operation on its local group tuple space. The template contains its identifier and the *sessionId* it offered.

4. Once the provider gets the session-tuple and accepts it, it runs an instance of the *psid* service, which becomes the *sessionId* session.

5. *ProviderPeer* can now confirm to *InvokerPeer* that its session is started by sending a session-ack-tuple in its local group tuple space. This tuple contains the invoker's identifier, the *sessionId* and a boolean.

[16] We assume this service to be **session-full**.

6. *InvokerPeer* calls `invoke(sessionId,opName)` which firstly puts an invocation-tuple into *ProviderPeer*'s local *groupId* tuple space. This tuple contains the operation name (`opName`) and its parameters (`input`), the service identifier (here the *sessionId*) and the identifier of the caller.

7. Since `invoke()`'s execution must be blocked until the provider has done a `receiveMessage()`, *InvokerPeer* will perform a (blocking) `in()` operation, waiting for the reply-tuple related to the `opName` operation.

8. When `receiveMessage(groupId,opName)` is called by *sessionId*, it retrieves an invocation-tuple from the local tuple space of its container (*ProviderPeer*) by doing an `in()` operation on it. The template of this operation contains only the operation name (`opName`).

9. Here the service instance (i.e. the session) actually executes the operation.

10. *SessionId* calls `reply()` which puts a reply-tuple into the local *groupId* tuple space of *InvokerPeer* (`opName` caller). This tuple contains the operation name, the caller identifier, the operation result and a possible fault. This last action will unblock the invoker's execution.

Event management

There are four primitives in event management: `event()`, `receiveEvent()`, `subscribe()` and `unsubscribe()`. The first two involve the release and reading of event-tuples in tuple spaces, while the last two have a Java-based only implementation.

Event

`void event(groupId?, eventName, input?) throws exception invalidGroupId, caller-NotInGroup, invalidEvent`

Primitive description. Entities call `event()` to raise an `eventName` event. Some data can be associated with it by using the `input` parameter. The `groupId` parameter, if specified, can restrict the visibility of the published event. If the group is not specified, the event is published in every group to which the caller peer belongs, in case the primitive is called by a peer. If the caller is a service, the event is released in the service's group. The SMEPP model requires the event to be published in the service contract in case of a service call (it would raise an *invalidEvent* exception otherwise). However, when we implemented our API, the definition of the service contract was not mature enough to allow such a verification, i.e. no event section was available.

Regarding the two other exceptions, it is obvious that *invalidGroupId* is raised if no `groupId` can be found and *callerNotInGroup* is raised if an entity wants to release an event in a group to which it does not belong.

Coordination view. Basically, an event publication is done by using an `out()` operation on the local group tuple space(s), which releases an event-tuple. This tuple is detailed in Figure 4.15. Nevertheless, to ensure the freshness of each event, `event()` deletes the (potential) previous `eventName` event (located in the caller's local group tuple space). Also, after releasing the event-tuple, a thread is started. It waits a few seconds and then deletes the event-tuple, this ensures the "factual" property of the event.

Implementation details. Since several entities share the same local group tuple space[17], one has to prevent an entity to delete an event published by another one. This is done by using the field (of the event-tuple) containing the `id` of the caller in the `inp()` template.

Exceptions handling. The two exceptions (*callerNotIngroup* and *invalidGroupId*) are raised in the usual cases. See above for more details.

[17] The services published in a same group share the local group tuple space of their containing peer.

<'_event',name,input,callerId,#,peerPwd>		
where	name	is the event name,
	input	is the data associated with the event (optional),
	callerId	is the sender identifier,
	#	means no read-password is used,
	peerPwd	is the remove-password generated by the peer.

Figure 4.15: Event-tuple.

ReceiveEvent

`<callerId, input?> receiveEvent(groupId?, eventName) throws exception invalidGroupId, callerNotInGroup, invalidInputParameter`

where `callerId` is either `peerServiceId`,`sessionId` or `peerId`.

Primitive description. `ReceiveEvent()` is similar to `receiveMessage()`. Its purpose is to retrieve an `eventName` event from the `groupId` group, optionally specified. If the group parameter is not provided, the primitive searches in all groups to which the entity belongs[18]. The primitive returns the event content (`input`) and the `callerId` of the provider.

Note that the call to `receiveEvent()` implies a previous subscription to `eventName`.

Coordination view. The implementation of `receiveEvent()` uses the SecureLime reaction. A `WeakReaction` is added to every group tuple space concerned (depending whether the `groupId` parameter is specified or not). This reaction reacts to a tuple (released in the tuple space to which it belongs) which matches the event-template (containing the `eventName`). The primitive then loops until the `EventListener` gets the expected event-tuple (Figure 4.15).

Once the reaction is triggered, it sets the newly published tuple inside the `EventListener` which allows the primitive to finally get unblocked and access to the event-tuple.

Implementation details. As with `receiveMessage()` the primitive has to behave differently if it is called inside a `Pick` or an `InformationHandler` command. However, the problem is simpler than in the `receiveMessage()` case, since the tuple is only read (and not deleted from the tuple space). If the primitive is executed inside such commands, it has to be possible for it to be stopped before it gets the event. Indeed, the branches of a `Pick` must be terminated when one of them is taken and the branches of an `InformationHandler` must terminate when the main command terminates. Thus, the implementation is made in such a way that the loop waiting for the `EventListener` to be full can also end if the parent command is terminated.

Exceptions handling. As in `receiveMessage()`, no check is made concerning the input parameter format. However the two other exceptions are raised in the usual cases.

Subscribe

`void subscribe(eventName?, groupId?) throws exception invalidGroupId, callerNotInGroup`

Primitive description. Entities call `subscribe()` to get subscribed to events. Depending on which parameters are specified, an entity can register itself to:

- every `eventName` raised in `groupId`, or
- every `eventName` raised in any group the caller belongs to, or
- every event raised in `groupId`, or
- every event raised in any group.

[18] If the container is a service, we assume the service belongs to the group it has been published in.

The two throwable exceptions are similar to the previous primitives.

Implementation details. The subscription to an event is internally handled inside an entity. Each entity uses its attached `EventManager` to manage the subscribed events. This management is made complex because of the flexible way an entity can register and unregister itself to events. For instance, a peer can subscribe to all events in any groups and later unsubscribe itself to an `eventName` event. Since available events are not published anywhere, one has to use a list of events and so-called anti-events. Then, when an entity wants to subscribe to all events, one adds a new special "all-event" to the list. After, if it unsuscribes to an `eventName` event, one adds an "anti-event" corresponding to `eventName` in the list.

Exceptions handling. The two exceptions are raised in the usual cases.

Unsubscribe

`void unsubscribe(eventName?, groupId?) throws exception invalidGroupId, callerNotInGroup, notSubscribed`

Primitive description. `Unsubscribe()` is the dual of `subscribe()`. Entities call this primitive to unregister themselves as event listeners. A call to this primitive can cancel, partially or not, a subscription.

Implementation details. The implementation of this primitive is similar to `subscribe()`. It adds or removes event and anti-event in/from the `EventManager` attached to the caller entity.

Event management scenario

This scenario illustrates how peers interact to publish and receive events. We assume two peers are part of the SMEPP application: *InvokerPeer* and *ProviderPeer*. The first peer subscribes to an event, which is released by the second one. Here are the SMEPP primitives executed by the peers:

InvokerPeer	*ProviderPeer*
`subscribe(evName,groupId);` `receiveEvent(groupId,evName);`	`event(groupId,evName,'someInput');`

Figure 4.16 shows that both peers are members of the *groupId* group since they have the *groupId* tuple space, the reference-tuple and the membership-tuple.

1. *InvokerPeer* begins by subscribing to the `evName` event. This is done internally in the peer.

2. The peer continues by calling `receiveEvent(groupId,evName)`. This will add a new `WeakReaction` on the `groupId` tuple space. This reaction reacts to an event template containing the event name. When the reaction is triggered, it executes the body of `evLis1`, an `EventListener` object.

3. Then, *InvokerPeer* waits `evList1` to be filled with an event-tuple received from the reaction.

4. *ProviderPeer* releases an event by calling `event()`. The added tuple contains the event name, the caller identifier and some associated data (`someInput`).

5. This last operation triggers the reaction added by *InvokerPeer*, the body of `evLis1` is executed and unlocks the execution of `receiveEvent()` by adding a tuple inside the `EventListener` object.

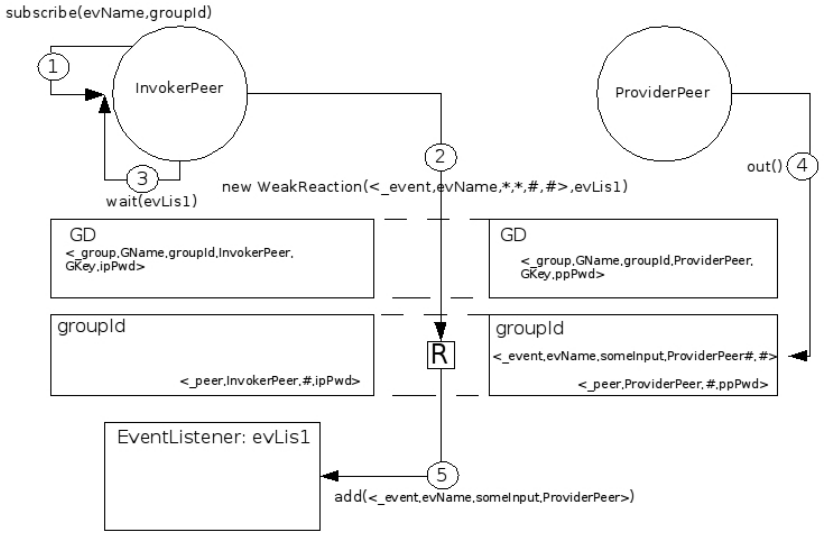


Figure 4.16: Service invocation.

4.3 Practical use

This section intends to present how the implementation actually works. Firstly, we introduce the reader to the steps to follow to use the software. Then, we illustrate the practical use by an example. Each step, from the translation to the executable program, is detailed.

4.3.1 General information

The result of our work consists in a Java "jar" file. This file has two purposes. Firstly, it is used to generate the Java code from a SMoL file. Secondly, once the Java files are compiled, the jar file is used to launch the newly created peers.

Basically, four steps are required to obtain a running SMoL peer (and its possible services). The procedure is as follows.

1. Write a SMoL code describing the behaviour of a peer with any text editor. This results in a XML file. If the peer has services, each service has to be written in a separate file.

2. Run the jar file to generate the Java code. In a command prompt, one must use:
   `java -jar SMoL2Java.jar -p <input_file.smol>`.

   This command outputs several Java files, the number of files depends on the complexity of the SMoL file.

3. Edit and compile the Java files. The programmer can here add some code in the generated files. Furthermore, the programmer must fulfil the boolean conditions of all the `While`, `RepeatUntil` and `If-Then-Else` constructs. To compile, one must use:
   `javac -cp SMoL2Java.jar *.java`.

4. Launch the peer. Once the `.class` files have been generated by the Java compiler, the user can launch the peer with:
   `java -jar SMoL2Java -v <port_number> <input_file>RootFH`[19]. SecureLime sends coordination information by using UDP multicast on the specified `port_number`. Note that SecureLime requires the port numbers to be separated by 3, at least.

[19]Note that <input_file>RootFH is the name of a Java class. The file name is the combination of the input file name and the string "RootFH", standing for "Root FaultHandler".

```
Sequence
   myCredentials = ⟨(SMEPPkey,SMEPPpwd),(TempReaderGroup,MyGroupNamePassword)⟩
   newPeer(myCredentials)
   myGroupDescription = <TempReaderGroup>
   tempGroupId = createGroup(myGroupDescription)
   Flow
      While true
         Sequence
            temp5s = ⟨opaque⟩
            event(tempGroupId, ‘‘temp5s’’, temp5s)
            wait(PT5S)
         End Sequence
      End While
      While true
         Sequence
            temp10s = ⟨opaque⟩
            event(tempGroupId, ‘‘temp10s’’, temp10s)
            wait(PT10S)
         End Sequence
      End While
   End Flow
End Sequence
```

Figure 4.17: TempReaderPeer's code.

4.3.2 Example

In this example, three peers interact. They all have the same credentials: one key to access the SMEPP application and one key to access the group called `TempReaderGroup`. The `TempReaderPeer` reads a temperature and publish the measurement via two event types (at intervals of 5 and 10 seconds). Those events are raised in a group created by `TempReaderPeer`. The first client (`ClientPeer1`) monitors the temperature through the "every 5 seconds" event. The second client (`ClientPeer2`) publishes a service (`ClientService`) which monitors the temperature during 30 seconds through the "every 10 seconds" events.

Step 1: Peers modelling

As explained above, the first step of the creation of a SMEPP peer consists in the writing of its code. Thus, we begin our example by giving the code of the three peers. We chose to present the code in a high level manner. However, the XML code is available in Appendix B.1.

TempReaderPeer. Figure 4.17 gives the code of `TempReaderPeer`. The peer starts by calling `newPeer()` with its credentials. Then, it creates the `TempReaderGroup`. The core of the peer's code is a two branches `Flow`. The first branch reads a temperature[20] and publishes it at interval of 5 seconds. The other branch does the same at interval of 10 seconds. The temperatures are published via events raised in `TempReaderGroup`. The complete code, written in XML is available in Appendix B.1.1.

ClientPeer1. Figure 4.18 gives the code of `ClientPeer1`. The two first statements are similar to the first peer. However, this client searches for the group `TempReaderGroup` by using the `getGroups()` primitive. Once the peer has the group identifier, it joins it. Then, it subscribes to the event corresponding to the temperature at interval of 5 seconds. The main part of the code is an `InformationHandler`. Its main command waits for one hour. Meanwhile, the unique branch waits for events (of the type "temp5s"). At each event reception, the peer "does something" with the event. This step is deliberately not specified in SMoL. The XML version of the code is available in Appendix B.1.2.

[20] This could require access to some hardware. Thus, it is not specified in SMoL.

```
1   Sequence
      myCredentials = ⟨(SMEPPkey,SMEPPpwd),(TempReaderGroup,MyGroupNamePassword)⟩
      newPeer(myCredentials)
      desiredGroupDescription = <TempReaderGroup>
5     gid[] = getGroups(desiredGroupDescription)
      join(gid[0],myCredentials)
      subscribe(''temp5s'', gid[0])
      InformationHandler
        Sequence
10        wait(PT1H)
          unsubscribe()
        End Sequence
        <cid, temp> = receiveEvent(gid[0], temp5s)
          ⟨use the event for something⟩
15    End InformationHandler
    End Sequence
```

Figure 4.18: ClientPeer1's code.

```
1   Sequence
      myCredentials = ⟨(SMEPPkey,SMEPPpwd),(TempReaderGroup,MyGroupNamePassword)⟩
      newPeer(myCredentials)
      desiredGroupDescription = <TempReaderGroup>
5     gid[] = getGroups(desiredGroupDescription)
      join(gid[0],myCredentials)
      clientServiceContract = ⟨ClientServiceContract⟩
      <gsid, psid> = publish(gid[0], clientServiceContract)
      wait(PT30S)
10    invoke(psid, monitor, gid[0])
    End Sequence
```

Figure 4.19: ClientPeer2's code.

ClientPeer2. The code of `ClientPeer2` corresponds to Figure 4.19. The behaviour is simple. Likewise in `ClientPeer1`'s code, the peer starts by joining the SMEPP application and, then, the `TempReaderGroup`. Once it has become a group member, the peer publishes a service, *ClientServiceContract*. Then, it waits during 30 seconds before invoking an operation exhibited by the service it just published. See Appendix B.1.3 for the XML version of the code. The signature of the service is showed in Figure 4.20.

ClientService. Figure 4.21 gives the code of the service published by `ClientPeer2`. The service starts by retrieving an invocation message corresponding to the "monitor" operation. Then, the service subscribes to the "temp10s" events which are raised in the group in which the service is published. The core of the service is an `InformationHandler` with one branch. Its main command waits during 30 seconds, then it cancels the subscription to the event. The only branch waits for "temp10s" events and, likewise in `ClientPeer1`, uses the event data for something. When the `InformationHandler` execution terminates, the service replies to the "monitor" invocation providing some data. The XML version of the service is available in Appendix B.1.4.

```
              serviceName   =  ClientService
              serviceType   =  session-less
request-response operation  =  monitor(groupId)
```

Figure 4.20: ClientService's signature.

```
Sequence
   <caller, input> = receiveMessage(monitor)
   gid = getPublishingGroup()
   subscribe(‘‘temp10s’’, gid)
   InformationHandler
      Sequence
         wait(PT30S)
         unsubscribe(‘‘temp10s’’,gid)
      End Sequence
      <cid, temp> = receiveEvent(gid, temp10s)
         ⟨use temp for something⟩
   End InformationHandler
   reply(caller,monitor,someData)
End Sequence
```

Figure 4.21: ClientService's code.

Step 2: Java code generation

In order to generate the Java files, the following commands have been used:

```
java -jar SMoL2Java.jar -p tempreaderpeer.smol
java -jar SMoL2Java.jar -p clientpeer1.smol
java -jar SMoL2Java.jar -p clientpeer2.smol
```

Note that the application finds itself the XML file representing the service contracts referenced in the peers. Indeed, the name of the file is required in the representation of the `publish()` primitive.

This step outputs the following files:

- **TempReaderPeer files:**
  `TempReaderPeerBRootFH.java`: The `FaultHandler`[21] which is the root of the code.
  `TempReaderPeerBDataTable.java`: The class containing the variables used in `TempReaderPeer`.
  `TempReaderPeerBFlow11.java`: The first branch of the `Flow`.
  `TempReaderPeerBFlow12.java`: The second branch of the `Flow`.
- **ClientPeer1 files:**
  `ClientPeer1BRootFH.java`: The `FaultHandler` which is the root of the code.
  `ClientPeer1BDataTable.java`: The class containing the variables used in `ClientPeer1`.
  `ClientPeer1BIH11.java`: The class representing the `InformationHandler`.
  `ClientPeer1BIH11Branch1.java`: The `receiveEvent()` branch of the `InformationHandler`.
  `ClientPeer1BIH11Branch1Cmd.java`: The command associated with the branch.
- **ClientPeer2 files:**
  `ClientPeer2BRootFH.java`: The `FaultHandler` which is the root of the peer code.
  `ClientPeer2BDataTable.java`: The class containing the variables used in `ClientPeer2`.
  `ClientServiceRootFH.java`: The `FaultHandler` which is the root of the service code.
  `ClientServiceDataTable.java`: The class containing the variables used in the service.
  `ClientServiceIH11.java`: The main class representing the service's `InformationHandler`.
  `ClientServiceIH11Branch1.java`: The `receiveEvent()` branch.
  `ClientServiceIH11Branch1Cmd.java`: The command associated with the branch.

All the files are given in Appendix B.2. In the following, we show some pieces of them in order to illustrate the code generation.

[21] Note that all `FaultHandler`s are implicit in this example.

Headers. Each file imports some packages from the SMEPP middlware API. For instance, `TempReaderPeerRootFH.java` begins with

```
1   import java.util.Vector;
    import java.util.Iterator;
    import peer.*;
    import peer.managers.services.*;
5   import sMoLTranslated.*;
    import utils.*;
    import utils.exceptions.*;
    import utils.ids.*;

10  public class TempReaderPeerBRootFH extends FaultHandler{ ... }
```

The two first imports are typical Java ones, while the others import classes from the API we developed. Furthermore, one can see that the object defined here extends the `FaultHandler` abstract class.

Body. The main part of a generated file is the `execute()` method. It contains the actual command of a construction. For instance, these following lines are the translation of `TempReaderPeer`:

```
1   public void execute(ThreadedCommand parent) {
      Pair tempPair = null;
      TempReaderPeerBDataTable.myCredentials =
        new Credential(new SMEPPKey("SMEPPKEY","SMEPPPWD"),
5       new SMEPPKey[]{new SMEPPKey("TempReaderGroup","MyGroupNamePassword")});
      peer.Primitives.getPrimitives().newPeer(TempReaderPeerBDataTable.myCredentials,instance);
      TempReaderPeerBDataTable.tempGroupId =
        peer.Primitives.getPrimitives().
          createGroup(new GroupDescription("TempReaderGroup"),instance);
10    TempReaderPeerBFlow11 TempReaderPeerBfl11 = new TempReaderPeerBFlow11(instance);
      TempReaderPeerBfl11.start();
      TempReaderPeerBFlow12 TempReaderPeerBfl12 = new TempReaderPeerBFlow12(instance);
      TempReaderPeerBfl12.start();
      try{TempReaderPeerBfl11.join();}
15    catch(InterruptedException e){e.printStackTrace();}
      try{TempReaderPeerBfl12.join();}
      catch(InterruptedException e){e.printStackTrace();}
    }
```

In the 6^{th} line, one can see the call to the `newPeer()` primitive. The first parameter is a variable representing the credentials and the second is a reference to the `FaultHandler`. This reference is used inside the primitive implementation to forward faults. In the same vein, the 9^{th} line is the translation of `createGroup()`. The lines from 10 to 17 are the statements used to create the different branches of the `Flow`. In this case, it starts two new Java threads, which are defined in `TempReaderPeerBFlow11` and `TempReaderBFlow12`. When the child threads are started, the main one waits for them using the Java's `join()`.

FaultHandler. In this example, each `FaultHandler` object contains the following lines at the end.

```
1   synchronized public void forwardFault(Fault f) {
      super.setStopped(true);
      boolean catched = false;
      Iterator it = super.getCatches().iterator();
5     if(!catched && it.hasNext()){
        catched=true;
        Catch c = (Catch) it.next();
        if(c.isAll() || c.getFault().getName().equals(f.getName())){
          System.out.println(f.getName()+" caught");
```

```
10            //catch main command
              System.exit(1);
           }
        }
     }
```

The purpose of this method is to handle faults. In this case, there are no `catch`es but only a `catchAll` clause. However, one can see the basic idea of the method. Firstly, it stops the execution of the command by calling `setStopped()`[22]. Then, it obtains the list of `catch`es and parses it.

InformationHandler. A particularly complex construct is the `InformationHandler`. Two `InformationHandler`s are present in the example: one in `ClientPeer1` and one in `ServiceClient`. They are both similar to each other. Thus, we can only focus on `ClientPeer1`. As explained in Section 4.1.1, the construct is translated to three objects[23]. The first file (`ClientPeer1BIH11.java`) contains the main command. Here, it consists in these lines:

```
1    public void execute(ThreadedCommand parent) {
        Pair tempPair = null;
        ClientPeer1BIH11Branch1 ClientPeer1Bih11b1 = new ClientPeer1BIH11Branch1(instance);
        ClientPeer1Bih11b1.start();
5       try{Thread.sleep(parser.SMoLParser.translateXPathDuration("PT1H"));}
        catch(InterruptedException e){e.printStackTrace();}
        peer.Primitives.getPrimitives().unsubscribe(instance);
        super.setFinished();
     }
```

In a few words, the method starts a new thread which corresponds to the (only) branch of the `Informationhandler`. Then, it executes the main command: it waits before calling `unsubscribe()` (lines 5 to 7). Finally, the method calls `setFinished()` to signal the end of the main command's execution.

The newly launched thread is defined in `ClientPeer1BIH11Branch1.java`. It executes the following lines.

```
1    public void run(){
        Pair tempPair = null;
        while(!instance.isFinished() || !instance.isStopped()){
           tempPair = peer.Primitives.getPrimitives().
5             receiveEvent(ClientPeer1BDataTable.gidArray[0],"temp5s",instance);
           if(instance.isStopped()) return;
           ClientPeer1BDataTable.temp10s = (Input) tempPair.getSecond();
           ClientPeer1BDataTable.caller_event = (CallerID) tempPair.getFirst();
           if(instance.isFinished() || instance.isStopped()) return;
10         ClientPeer1BIH11Branch1Cmd ClientPeer1Bih11b1cmd =
              new ClientPeer1BIH11Branch1Cmd(instance);
           ClientPeer1Bih11b1cmd.start();
        }
     }
```

This executes the `receiveEvent()` primitive while the main command is still running and no faults have been raised. In order to avoid to execute the associated command if a fault has been raised[24], one needs to check it before starting the corresponding thread. In case a fault has been raised (`isStopped()`) or the main command is finished (`isFinished()`), the method terminates its execution prematurely via a call to `return` (see lines 6 and 9).

The associated command of the branch is defined in `ClientPeer1BIH11Branch1Cmd.java`. This file contains only a `run()` method which is responsible of executing the translation of the command. In this example, no SMoL code was associated with the branch, so the method body is empty.

[22] The command's children check the `stopped` variable regularly.
[23] Actually, if an `InformationHandler` has n branches, it is translated to $n+1$ objects.
[24] Inside `receiveEvent()`, for instance.

Step 3: Editing and compiling

At this stage, the programmer can add some platform-dependant code into the generated files. In this example, we only add a Java `while` before the calls to `getGroups()`. This avoids the primitive to return `null` if the group creator is not running yet. In this way, one can run the peers in any order.

In the root file of Client Peer 1 (see Appendix B.2.2), we added those lines around the `getGroups()` primitive.

```
int i = 0;
while(i==0){
   ClientPeer1BDataTable.gidArray =
      peer.Primitives.getPrimitives().
         getGroups(new GroupDescription("TempReaderGroup"),instance);
   i = ClientPeer1BDataTable.gidArray.length;
   try Thread.sleep(1000); catch (InterruptedException e) e.printStackTrace();
}
```

This additional code forces the peer to call `getGroups()` until it returns an array containing at least one element. To avoid a high CPU load, the thread sleeps for 1 second between each call.

Once the programmer has done all the changes he wanted to, he can compile the files with the Java compiler. This is typically done with "`javac -cp SMoL2Java.jar *.java`". This compiles all the Java files in the current directory. The `-cp` parameter ensures the compiler has access to the objects declared in the jar file.

Step 4: Running the peers

The last step consists in running the peers. In this example, we use the following three commands.

```
java -jar SMoL2Java.jar -v 5555 TempReaderPeerBRootFH
java -jar SMoL2Java.jar -v 6666 ClientPeer1BRootFH
java -jar SMoL2Java.jar -v 7777 ClientPeer2BRootFH
```

The `-v` parameter enables the verbose mode in which every primitive displays some information about its execution. Three screen captures are available in Appendix B.3. The first capture shows the behaviour of `TempReaderPeer`. After its initialisation, the peer sends events at regular intervals. The second capture illustrates `ClientPeer1`. After discovering the groups and joining `TempReaderGroup`, it waits for the "temp5s" event. The last capture is related to the second client. After joining the group, it published a service in it. Then, the capture shows the executions of the peer and its service. One sees the peer invokes the "monitor" operation, the service receives the corresponding message and starts its execution. Basically, the service listens to the "temp10s" event for a while and, then, replies to the caller. One can see, in one of the last lines, that the peer receives the dummy text "`anyOutputPossible`". In a real application, it could be any value (for instance, the average of the received events values).

4.3.3 Implementation tests

A series of tests have been performed to ensure that the behaviour of the proof-of-concept meets the SMEPP specification. The following steps have been achieved successfully on a Linux machine.

- The translation and execution of each SMoL command (from a XML file to a Java program).
- The execution of all primitives, independently.
- A simple SMoL application, featuring two SMEPP peers and a service, presented in the paper we published (see Appendix C).

- The aforementioned example was used to ensure a whole SMoL application works properly, from the XML file to the Java executable code. Remember this example features three SMEPP peers and a service as well as "complex" SMoL commands.

Unfortunately, due to the lack of a tool to assist the development of SMoL programs, testing more complex applications is very time-consuming (please see the XML codes of the aforementioned example, in Appendix B.1). We believe more tests could be useful. Nevertheless, the set of tests we have achieved give us confidence in the compliance of our application with the SMEPP service model.

Chapter 5

Perspectives

The previous chapter showed that the proof-of-concept implementation meets its goals. However, the implementation context is very restrictive. Indeed, the implementation is built on top of the SecureLime API which brings its own limitations. In the same vein, the SMEPP service model is only theoretically defined and its implementation raises many questions which are not always easy to address. Thus, some compromises had to be made. In some cases, they constitute questionable weaknesses of the prototype.

This chapter brings a critical point of view on our proof-of-concept implementation. This look shows the compromises we made, classifying them according to their relations with the implementation context. Firstly, Section 5.1 gives the criticisms related to the SecureLime API, while the following one (Section 5.2) presents those related to the gap between the SMEPP service model and the tuple space based coordination model. Finally, Section 5.3 suggests future work which could be interesting to be investigated.

5.1 SecureLime related issues

This section presents the implementation weaknesses having as main origin the use of SecureLime as coordination system. We firstly discuss the lack of session keys mechanisms and the way this could be added to the implementation. Then, we analyse the context management limitations due to some restrictions on SecureLime's operations. Finally, we study the addition of timed primitives to the coordination language in order to enhance retrieval operations while staying compliant with SMoL specifications.

5.1.1 Keys refreshment

In the implementation, all the communications are protected using symmetric pre-shared keys (*AppKey* for groups and services discovery, *GKeys* for group communication). While this mechanism actually ensures data secrecy, it has a main flaw. Indeed, a well known security principle states that pre-shared keys should be used as sparingly as possible. In ad hoc network, communication channels are often not secure (e.g. wireless networks), thus an attacker can easily eavesdrop the communications. Furthermore, the more a key is used, the more chances an attacker has to guess it. Therefore, mechanisms to limit the amount of cipher-text available to cryptanalysis for the given key are needed. The standard solution is the use of session keys, which are refreshed regularly. The lack of such a mechanism is an important flaw in our implementation.

In SecureLime, once a protected tuple space is created, the password cannot be changed. Therefore, implementing a session keys mechanism would require every peer to do the following steps: backup all the tuples contained in its local tuple space, create a new one (using a new password) and put the saved tuples in the new tuple space, using `out()` operations. Furthermore, if reactions are registered in the tuple space, they must be registered again in the new one. Besides its obvious lack of efficiency, this mechanism would cause consistency problems

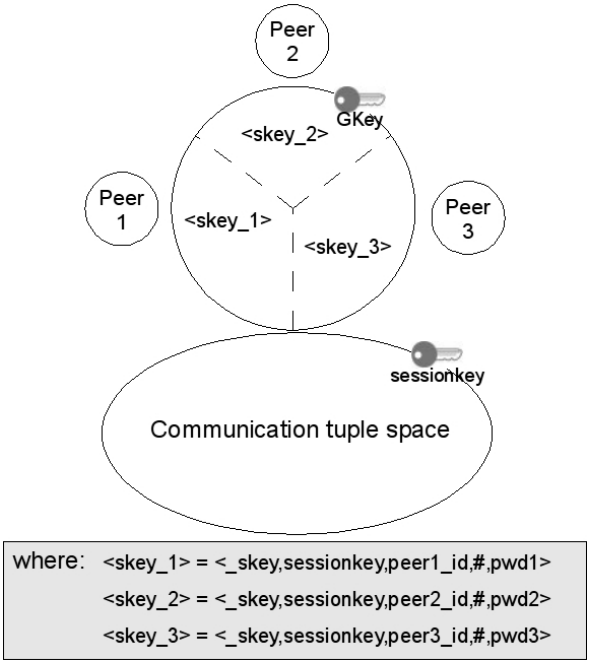


Figure 5.1: New design.

between the two tuple space versions, in particular when a peer invokes an operation, i.e. tuples move from a local tuple space to another. Moreover, the re-registering of reactions could imply the loss of some events.

In order to implement a working session keys mechanism, we need a coordination system which adds a mean to change on-the-fly the key protecting a tuple space. In SecureLime terms, this could be modelled as the tuple space disengagement, the key update and the tuple space re-engagement. Assuming the availability of a new SecureLime version, featuring on-the-fly password changing, we propose the following design.

Each group is modelled into two tuple spaces. The first is protected using the corresponding *GKey* and serves to exchange the session keys which protect the second tuple space. This one, called communication tuple space in the following, is used in the same way as in our previous design (see Section 4.2.1).

Figure 5.1 illustrates the new design. Three peers are members of a group which is represented as two tuple spaces. The top of the figure shows the tuple space used to exchange and refresh the keys. It is protected by *GKey* and every peer has put a tuple containing the current session key (see hereafter) in its local tuple space. The middle of the figure represents the tuple space used for other interactions.

The new design requires to review group creation and joining. In order to create a new group, a peer performs the following steps:

1. It creates a new tuple space using the group's identifier and the corresponding *GKey*.

2. It inserts a tuple into this new tuple space. This tuple has the form:
   <_`'skey',sessionkey`$_i$`,tuple_owner_id,#,remove-pwd` > (called skey-tuple).

   The first field is simply a tag, the second one contains a securely generated session key and `tuple_owner_id` is the peer's identifier. Moreover, the tuple is remove-protected by a password kept secret by the peer.

3. The peer registers a reaction on the tuple space which reacts to the template:
   <`'_refresh',initiator_id,#,remove-pwd`> (called refresh-tuple)[1].

[1]If `initiator_id` is the identifier of the peer, the reaction's body is aborted.

When fired, it reads the skey-tuple where `tuple_owner_id = initiator_id`. Then, it updates its own skey-tuple by changing the `sessionkey` field. The reaction terminates by changing the key of the communication tuple space.

4. Finally, the peer creates a second tuple space using the group's identifier and the session key it has chosen in the second step. This tuple space is used for the actual group communications.

When a peer wants to join a group, it executes the following steps:

1. It creates a new tuple space using the group's identifier and the corresponding *GKey*. This tuple space merges with other tuple spaces having the same name and key.

2. It reads a skey-tuple from the tuple space and extracts the session key contained in it.

3. Then, it creates a skey-tuple where the `sessionkey` is the previously extracted key, `tuple_owner` is its own identifier and `remove-pwd` is a chosen secret password.

4. It creates the same reaction as in step 3 of the group creation.

5. Finally, it creates a new tuple space using the group's identifier and the session key it read before.

Furthermore, we add a mechanism to refresh keys. A peer initiates a key refresh when it notices that it has used the same key for a certain amount of time or of data[2]. Here are the steps involved in the refresh procedure when triggered:

1. The initiator peer starts by changing the key of its communication tuple space. We assume the key is securely generated.

2. It updates its skey-tuple with the newly generated key.

3. It inserts a refresh-tuple, protected by a chosen password, containing its own identifier. It also starts a new process which will delete this tuple after a few seconds[3].

4. The refresh-tuple triggers the reaction registered by other members at group joining/creation. Thus, they update their skey-tuple and change the key of their communication tuple space.

5. After that, every member uses the same key.

The peer creation has also to be reviewed, in particular the way the application level tuple spaces (*SD* and *GD*) are protected. In concrete terms, one applies the same mechanism to the *SD* and *GD* tuple spaces. This requires an additional tuple space, protected by the application key (*AppKey*), used to exchange and refresh the keys. *SD* and *GD* keep their respective roles but are now protected by the session key established in the first tuple space. Thus, the new version of `newPeer()` is similar to create and join a group. When a peer engages the tuple space protected by *AppKey*, it is not sure it will find a skey-tuple, since it may be the first peer to join the network. In this case, it is similar to the group creation: the peer generates a key, publishes it in a skey-tuple and creates the discovery tuple spaces. If the peer is not the first one, it is similar to join a group: the peer reads a skey-tuple and then creates *SD* and *GD* with the newly read key.

[2] One could use either a timer or a variable counting the number of exchanged tuples.

[3] Note that if the tuple stays in the tuple space, it would trigger unnecessarily the reaction registered by new members.

5.1.2 Context management limitations

As explained in Section 3.5, SecureLime enforces to specify the location parameters when using probe (`inp()` and `rdp()`) and aggregate (`rdg()` and `ing()`) tuple space operations. However, the needs to read the whole tuple space in a non-blocking manner and to retrieve multiple results are common to many applications. When one needs to use these operations on a federated tuple space, SecureLime requires to know all the identifiers of the connected peers.

To be able to invoke these operations on a federated tuple space, our implementation uses a workaround which retrieves a list of the peers' identifiers from the `LimeSystemTupleSpace` and, then, loops on it, invoking the actual operation. Although the workaround is easy to implement, it creates new problems. Indeed, between the instant where the peers' identifiers are retrieved and the instant where the operation is called, several peers could have joined the network. This may lead to inconsistent results. One may miss a tuple added by a peer which has just joined the network.

Unfortunately, in the current implementation of SecureLime, it seems not possible to bypass this workaround. The authors of Lime justify this limitation, saying "a unconstrained definition, such as the one provided for in and rd, would involve a distributed transaction across the federated tuple space to preserve the semantics of the probe" [75].

While this problem is not critical for the implementation use, it highlights the lack of space uncoupling in SecureLime. Indeed, instead of querying a whole tuple space, the user is forced to know in which peer's local tuple space the data it is looking for is located.

However, space uncoupling is a feature, with time uncoupling, that made the success of tuple space based coordination system (please refer to Section 3.1.2). In addition, space and time uncoupling eases the context management because the peers do not have to synchronise to exchange information about their states. In the case of SecureLime, a peer cannot retrieve a global knowledge of the network without knowing its structure (i.e. the set of connected peers).

In [57], Herrmann compares Lime's network transparency with his MeshMDL middleware. "(...) Lime federates tuple spaces of neighboring nodes. While this creates some degree of transparency for the programmer by presenting only one space to him, it also increases the coupling of nodes. The space concept in Lime is much more complicated since there exist several spaces for agents and nodes that need to federate. Interestingly, the designers of Lime discovered that total transparency is not achievable. As a result they integrated operations into their space model that break transparency and let callers deal explicitly with individual spaces at specific locations."

5.1.3 Timed primitives

The SMoL specification defines that any command must be terminated as soon as a fault is raised inside the `FaultHandler` including it. This poses a problem when the implementation of a primitive requires to use a SecureLime blocking operation (such as `in()` and `rd()`). Indeed, as the execution is blocked, when a fault is raised, it is impossible to terminate the waiting. Thus, to comply with the SMoL specification, we never use any blocking operation. We implemented "active waiting" versions of these operations. Basically, they are implemented by a loop which iterates until a matching tuple is found or a fault is raised. Inside the loop, we invoke the operations' probe versions and, in case no tuple is found, we suspend (via a Java `sleep()` call) the execution of the thread for a few seconds.

This workaround has many flaws. Firstly, it is resource demanding since it may loop for a long time (or even forever). Secondly, the sleep period may lead to starvation issues. Assuming a loop sleeps for 1 second every cycle and an operation takes 1 millisecond, if the loop lasts 1 minute, a peer actually searches for ($\simeq$) 0.06 second. Therefore, the rest of the time can be exploited by concurrent operations and a caller could never get any result. Thirdly, the use of probe operation, if invoked on the whole federated tuple space, creates the same problem as discussed in the previous paragraph. Finally, the workaround may interfere with the coordination system's scheduling of operations. Although Lime and SecureLime documentations do not give any information on the fact that the operations are orderly fulfilled, one can imagine the system must be implemented in a deterministic way. However, in our implementation, the scheduling

is non-deterministic. For instance, if two peers successively invoke an `in()` operation with the same template[4], SecureLime may ensure that the first caller is the first to get a result. In our implementation, this does not stand anymore because of the sleep period. If a tuple appears during the sleep period of the first caller and the second one's invocation occurs during this period, the second caller gets a result before the first one.

In order to avoid these problems, it is necessary that the coordination system features timed operations. For instance, such a primitive may have the form: `inp(<template>,timeout)`. We expect the middleware to schedule the operations and to actually search for a `timeout` period (and not for a small percentage of it).

Note that since some primitives are blocking, it is still necessary to use these timed primitives inside a loop which regularly checks whether a fault has been raised or not.

Two of the systems surveyed in Section 3.3 provide transactional features and timed versions of the primitives such as the one previously described. These systems are JavaSpaces [48] and MARS [24]. However, these two languages do not meet the SMEPP peer-to-peer orientation requirement (`R1` in Section 2.5). Therefore, they cannot be used to implement our SMEPP middleware proof-of-concept.

5.2 SMEPP related issues

This section presents the issues due to the gap between the SMEPP model and our tuple space based implementation. Firstly, in order to highlight the problems they raise, we discuss the differences between the way events are modelled in SMEPP and in tuple space systems. Then, we analyse the choice made to make SMoL an executable language. Finally, we propose a new architecture design which allows to use the API without the SMoL to Java translator.

5.2.1 Event handling

The SMEPP service model manages events in a quite non-intuitive way. An entity must subscribe to an event before being allowed to receive it. Furthermore, to actually retrieve an event, it must call the `receiveEvent()` primitive. A more intuitive solution for event handling is the publish/subscribe model [58]. In this model, entities subscribe to events of their interest and receive them "automatically" when new events are raised. Thus, in this model, the receiver is passive after the subscription. No blocking operations are needed to receive an event, i.e. entities can continue their execution.

Similarly to other languages providing event modelling (such as MARS, KLAIM, LuCe, etc. [24, 41, 43, 48, 60, 71, 74, 84]), SecureLime only features the publish/subscribe model which is implemented via the "reactions". Therefore, the mapping from SMEPP events to Lime concepts is not obvious. The difference between the two models led us to an implementation where the subscription to a SMEPP event is only locally handled. The subscriber is the only one to know about its subscription. Thus, the subscription is only used to control the entities' rights to receive or not an event. It is when an entity invokes `receiveEvent()` that we register a new reaction on a tuple space. When, the reaction is triggered, it returns the event-tuple to the caller. This implementation does not take advantage of the SecureLime reactions. Indeed, the program is blocked to comply with the SMEPP specification. However, it could benefit from the reaction implementation by continuing its execution while still being given the event at reception time.

This problem makes us reconsider the joint use of the `subscribe()` and `receiveEvent()` primitives. We believe that one of them appears to be quite unnecessary in the tuple space model. Indeed, a subscription to an event is never refused but a call to `receiveEvent()` implies a previous subscription. Thus, it suffices to call `subscribe()` before `receiveEvent()` to avoid a refusal. Furthermore, the model does not specify any lifetime for events. This complicates the choice to be made in order to implement the release and reception of events. If several events of the same type are published, the issue is twofold. Firstly, is the existence of more than one event of the same type allowed? Secondly, if it is allowed, which event is

[4]The problem can also occur when the templates match a common set of tuples.

read by `receiveEvent()`? On the one hand, taking the `subscribe()` (and `unsubscribe()`) away from the service model would not affect the way it handles events. Furthermore, it would simplify the SMEPP middleware from the user point of view. The current implementation still fits this modification. It only requires to remove the subscription handling part. On the other hand, switching to the publish/subscribe model would allow to match directly the tuple space concepts. Furthermore, it would solve the issue related to the choice of the event to be retrieved. However, it would require to remove the `receiveEvent()` primitive. This change is much heavier than the previous one. Indeed, it would require to rethink all the event handling in SMEPP.

5.2.2 Target language dependence

Since SMoL is a specification language and our goal was to produce executable code, we had to adapt the concrete representation of the data. Indeed, SMoL programs handle many data which need to be modelled in Java. This implies that a SMoL code must contain all the needed information.

For instance, the representation of the credentials used in an application is not defined by the service model. However, the implementation requires to have the structure of a credential defined. The chosen representation[5] has many consequences on both the implementation and the way SMoL code has to be written. Since SecureLime uses password to protect the tuple spaces, the credentials we defined contains arrays of strings representing them. Thus, the SMoL programmer undergoes the security features of SecureLime. Assuming another implementation would use asymmetric keys based mechanisms to secure the communications, the same XML representation of credentials could not be used. SMEPP should define an abstract representation which must be generic enough to encompass all kinds of credentials.

Another similar issue is caused by the input and output parameters of service operations. The service model ensures that an operation invocation matches the types of input and output parameters defined in the signature. This poses a problem since the signature describes the parameters in an abstract manner and the implementation of the operation needs concrete data. For instance, in our implementation, operations are Java-based. Thus, the parameters must be mapped to Java. Two solutions are possible. The first solution defines every object used by a SMoL program into an abstract XML structure and defines a corresponding translation to Java. The second one uses directly the target language (here, Java) in the XML files. In order to illustrate these solutions, we start with the abstract `assign` command:

`Copy from` *5* `to` *temperature* `End-Copy`.

In the first solution, the "`from`" part must be written as:

`<from><literal><Integer><value>5</value></Integer></literal></from>`.

The flag `Integer` is used by a translation rule to produce the corresponding Java code. The "`to`" part is written as `<to variable=''temperature''/>`. This solution, although elegant, implies much work since it requires the implementers to think about all the possible data structures (or objects) and their translation. Furthermore, the way the structures are defined has a strong object-oriented connotation which may not be always wanted.

The second solution would model the "`from`" part as:

`<from><literal>new Integer(5)</literal></from>`.

One sees that some Java code is directly used inside the XML files. This solution is much easier to translate, but it wipes out the SMoL's intention to be independent towards the SMEPP middleware implementations.

Our implementation uses mainly the first solution. Many data structure are defined in the XML schema. However, a problem remains to reach fully the first solution. It concerns the input/output parameters of service operations. We could not find a way to define all the data structures for every kind of parameters. To bypass this limitation, we defined a generic data type, `Input`, which can contain any Java Object. This generic object is used similarly as the second solution proposed before (i.e. using the `Input`'s constructor).

[5] See Appendix A.2.

```
try
   {
      String tmp_variable = (String) invoke(peerServiceId,operationName,AnyInput);
   }
catch(ClassCastException e )
   {
      parent.forwardFault(new Fault(invalidOutputParameter));
   }
```

Figure 5.2: Output parameter type check.

The main drawback of this workaround is that it makes impossible to check that the type of input/output parameters of the `invoke()` and `reply()` primitives match the type defined in the operation's signature. To solve this problem, we need all possible data structures handled by service operations to be defined in abstract terms (i.e. in the XML schema). We also need a translation rule to the target language for every data structure. Furthermore, every input and output parameters used in operation signatures must refer to an existing definition.

In that case, the input parameters can be checked inside the `invoke()` implementation. It suffices to ensure that the concrete representations of the input data refer well to the signature. Otherwise, the primitive raises an `invalidInputParameter` exception. The same verification has to be made in `receiveMessage()` on the provider side.

In Java, checking the type of output parameters is slightly more complicated. It requires to encapsulate the actual call to `invoke()` in a `try/catch` block. Figure 5.2 shows how it is made. The operation output type is used to cast the generic object returned by `invoke()` in the signature expected type (here, `String`). In case Java raises a `ClassCastException`, this means the types do no match, thus an `invalidOutputParameter` exception is raised.

5.2.3 Translator and API coupling

As a late requirement of our work, we have been asked whether it is possible to use our API without using SMoL (i.e. without the translator). In the current state of the implementation, this requires additional work from the user. Indeed, the API and the translator are tightly coupled, mainly because of the SMoL fault handling, but also because of some commands (i.e. `InformationHandler` and `Pick`). The behaviour of the primitives is influenced by their uses inside such commands, which requires a synchronisation between them. For instance, concerning fault handling, the primitives have to interact with their parent `FaultHandlers` in order to stop other possibly running child `Flows`, if need be.

To use the API without the translator, the following is required. Since the API expects the executable code to be included inside a `FaultHandler`, the programmer has to write his program inside a "pseudo `FaultHandler`", as the one described in Figure 5.3. This ensures that when a SMoL fault is forwarded to it, it is translated to a Java `RuntimeException`. Running a SMEPP peer remains the same. One simply needs to compile and execute the pseudo `FaultHander` as usual (see the fourth step in Section 4.3.1).

Moreover, since SecureLime restricts access to tuple space[6] only to `LimeAgent` and `LimeThread` objects, the programmer cannot use the classic Java `Thread` class. Therefore, we provide a new abstract class which allows concurrent programming while keeping access to tuple spaces. This class is given in Figure 5.4. It implements `ILimeAgent` and extends `LimeThread` to comply with SecureLime requirements.

This is the easiest way to use the current implementation without the translator. Note that the programmer has still to provide the reference to the pseudo `FaultHandler` as a parameter to every primitive.

A more elegant solution would require to review a substantial part of the design. We propose in the following a new design which takes into account the desire to use the API without the

[6]Please remember that the primitives access tuple spaces.

```
1   public class PseudoFaultHandler extends FaultHandler {
       ThreadedCommand parent = null;
       public PseudoFaultHandler() {
          parent = this;
5      }
       public void execute(ThreadedCommand parent) {
          ⟨Peer's or service's body.⟩
       }
       synchronized public void forwardFault(Fault f) {
10        stopped = true;
          throw new RuntimeException(f.toString());
       }
       public boolean isStopped() {
          return false;
15     }
    }
```

Figure 5.3: Pseudo `FaultHandler`.

```
1   public class SMEPPThread extends LimeThread implements ILimeAgent {
       private Threadcommand parent = null;
       public SMEPPThread(ThreadedCommand cmd) {
          parent = cmd;
5      }
       public void run(){
          ⟨Thread's body.⟩
       }
       public LimeAgentMgr getMgr() {
10        return getParent().getMgr();
          //to comply with ILimeAgent interface
       }
    }
```

Figure 5.4: SMEPP Thread class.

translator. This design re-use a substantial part of the current one. It keeps the `Primitives` class which contains the implementations of the primitives and the three SMEPP entity classes (`Peer`, `Service` and `Session`) which keep their structure (e.g. groups, services managers etc.). Figure 5.5 gives the class diagram of the new design. In order to maintain the restrictions on primitives call[7] when the translator is not used, we divide the primitives into three classes, namely `SMEPPEntity`, `Peer` and `Service`. The first one contains the primitives available for both types of entities, in the form of *public* methods. The second one, `Peer`, features the peer specific primitives, as *private* methods. Finally, `Service` contains the only primitive available only to services (i.e. `getPublishingGroup()`, without parameters) as a *private* method.

It is important to mention that the actual implementations of the primitives are still in the `Primitives` class. Therefore, the methods in the three entity classes are wrapping them to hide the SMoL fault handling and primitive synchronisation features.

Remember that, in `Primitives`, the methods take as last parameter the SMoL command including them (i.e. a `ThreadedCommand` object). However, when SMoL is not used, this last parameter is not needed and is disturbing for the API user. To hide the SMoL related features, all primitives' implementations in `SMEPPEntity` and its children call the methods from `Primitives` specifying as last parameter their own reference[8]. Furthermore, the `forwardFault()` method is overridden to raise a Java exception instead of a SMoL fault. In addition, `isStopped()` is overridden to return always `false`, since it is only used in SMoL `FaultHandler`.

[7] Some primitives can only be called by peers, one can only be called by services and some can be called by both.

[8] As `SMEPPEntity` implements `ThreadedCommand`.

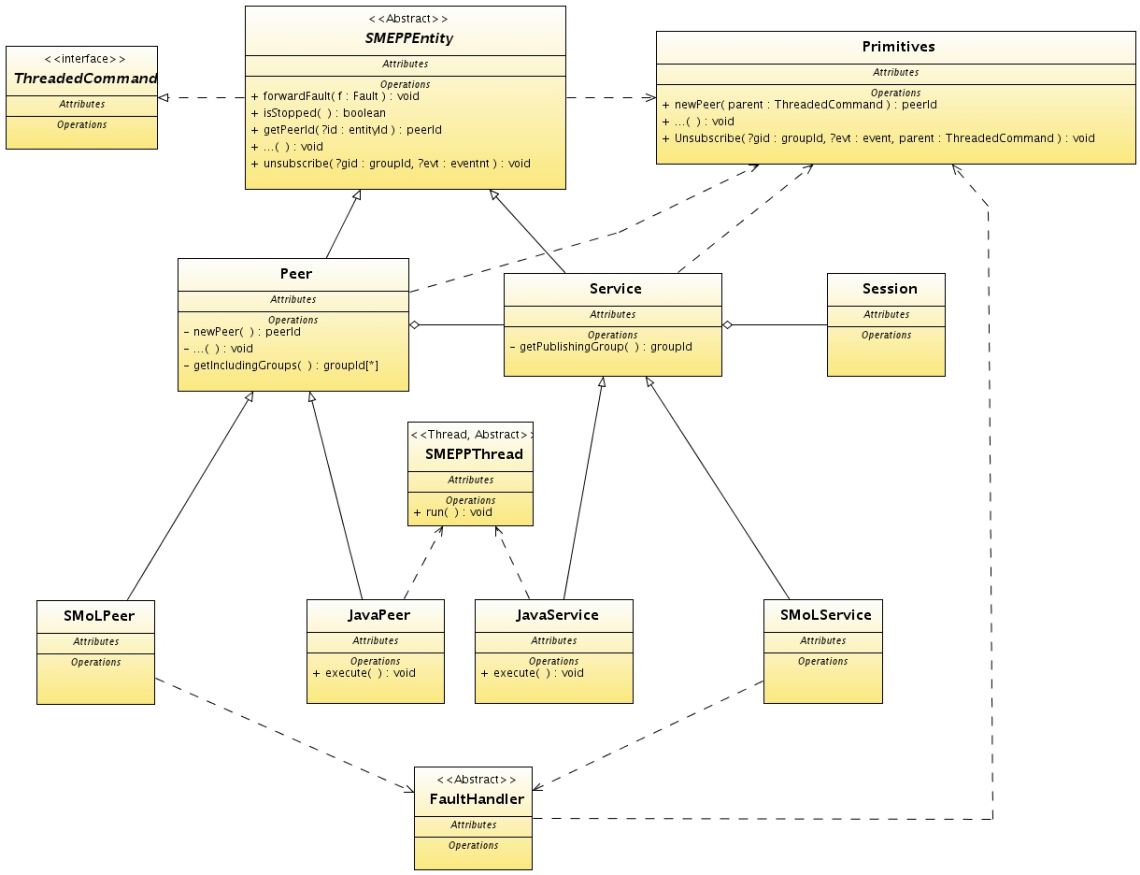


Figure 5.5: New design proposal.

The programmer wishing to create a SMEPP peer with the API, but without the translator, has to create a new class extending `JavaPeer`. In this class, the peer application code has to be written inside `execute()`. Every call to a primitive is implemented by a call to the primitive methods defined in `SMEPPEntity` or `Peer`. Similarly, when the programmer wants to create a service, s/he creates a new class extending `JavaService`. However, in this case, the methods available are the ones in the `SMEPPEntity` and `Service` classes. If multi-threading is needed, instead of using standard Java `Threads`, the programmer has to use our `SMEPPThread` class given in Figure 5.4.

When the translator is used, the way to proceed is the same as before. The translator generates a `FaultHandler` class for each peer and service. This class is the root of the entity's body. As in the current implementation, primitive calls in `FaultHandler` refer to the `Primitives` class directly.

The advantage of this solution is that it does not imply redundant implementations of the primitives (one for SMoL entities and one for Java ones). Furthermore, it does not require many changes of the current implementation. It comes quite well along on top of the current design. Above all, this solution does not require the programmer to know about SMoL if s/he does not want to. Unfortunately, as the requirement of using the API separately from SMoL came late in our implementation work, we have not been able to propose a concrete solution in time.

5.3 Future work

This section intends to outline new topics of work which could improve the architecture and implementation we proposed. Firstly, Subsection 5.3.1 proposes a new design which permits to meet SMEPP security *level 2* requirements. Secondly, Subsection 5.3.2 discusses new tuple matching policies which can ease the implementation of the service model. Then, Subsection 5.3.3 studies the feasibility of a similar proof-of-concept meeting interoperability requirements. Finally, Subsection 5.3.4 proposes a proof scheme to validate the middleware implementation and the SMoL translator.

5.3.1 Higher security level

The current implementation features only SMEPP security *level 1*. Remember that this level uses only symmetric keys. The main issue is that the security keys have to be pre-shared. Thus, after the application initialisation, it is impossible to exclude members from a group or to create non-foreseen groups. Furthermore, in security *level 1*, every peer may endanger the whole application if its credentials are leaked. In contrast, in security *level 2*, based on asymmetric keys, every entity has its own private/public keys couple. Therefore, if a credential is leaked, it only affects its owner. In such a case, the owner can be blacklisted, which has the consequence that the application preserves its security. The main challenge of *level 2* is to ensure such security guarantees without relying on a centralised entity inside the SMEPP applications.

Currently, *level 2* is not yet completely specified in SMEPP. However, it would notably extends the possibility of group creation by allowing to create dynamically new groups, without requiring all the peers to know a priori their existences. These "dynamic" groups can define a policy which specifies how the decision to accept a new member is taken and, possibly, how members decide to exclude another one.

In the following, we start by giving the assumptions which have to made concerning the SMEPP service model and the coordination system on top of which the implementation would be built. Then, we propose a design meeting *level 2*.

Assumptions

In order to propose a new design overview of a SMEPP proof-of-concept based on tuple space system, we need to define two types of assumptions. On the one hand, we have to detail on what basis the SMEPP security *level 2* can be built. On the other hand, we have to elicit the requirements the coordination system has to meet to be a suitable basis.

We propose the following extensions to the security related assumptions on SMEPP entities:

- each peer trusts a Certification Authority (CA) which delivers public key certificates,
- the interactions with the CA take place outside the SMEPP application,
- each peer has a couple of private and public keys,
- each peer owns a public key certificate which binds together its identity to its public key,
- each certificate incorporates the usual security features, such as the public key being signed, the peer identity, a validity period and the digital signature of the certificate provider. This ensures the peer's right to access the application,
- each certificate also contains a set of attributes consisting in a set of groups the owner can join,
- each peer owns a set of keys which defines the group categories it can see. These categories are used to rank the security level of groups, each category having a key used to restrict its visibility. For instance, one could have a 3-categories application, featuring *free*, *confidential* and *top-secret* groups. A peer having the keys corresponding to *free* and *confidential* can discover all the groups contained in these categories.

- services' visibility is defined by the group in which they are published[9],
- the set of categories the peer is allowed to see is contained in the peer's certificate,
- two group types are possible:
  - Static groups are similar to *level 1* ones, but rights to access them are defined in peers' certificates,
  - Dynamic groups do not need to be foreseen before runtime. Accessing them is restricted through group policies. The policy type is decided by its creator and can involve other members participation. The group creator is able to generate a private/public key couple for the group.

In conclusion, the application access level is restricted by the validity of the peer's certificate. The group visibility is restricted by the ownership of the corresponding category keys. Furthermore, group access depends on the group type. Static group authorisation is managed through the peer's certificate, while dynamic group authorisation is managed through group policies. Finally, services' visibility depends on the group category in which they are published. Note that peers can discover services they cannot invoke, because they may not have access to the group in which they are published.

To develop a tuple space based proof-of-concept of SMEPP which meets security *level 2*, one needs a suitable coordination system. We propose that it should have the following features:

- transiently shared tuple spaces, similar to Lime's ones,
- tuple level access control providing read- and remove-protections,
- matching policy similar to the SecSpace [52] one, based on asymmetric keys (see Section 3.3.1, p. 53),
- tuple space level access control, similar to SecureLime, but based on a stronger type of keys and which can be changed on-the-fly,
- programmability of the tuple space, similar to the Lime's reaction, and,
- a transactional version of `outg()` (as in Lime) which allows to insert several tuples in an atomic manner.

We assume that the tuple level access control uses the following matching policy. It is similar to the SecSpaces one but allows to leave unspecified the targeted peer in the *asymmetric partition field*. In SecSpaces, the matching rule provides for one-to-one communication. Using this new matching rule, one-to-many communication is possible. We model a tuple as follows: $<\texttt{field}_1,\ldots,\texttt{field}_n,\texttt{read-key},\texttt{remove-key}>$. In case of a `rd()`, a tuple and a template match only if their fields and their `read-key`s match. The matching rule is the same for `in()`, but the `remove-key`s must match too. The matching process for `read-` and `remove-key` fields is defined as follows. Two protection fields match if and only if one is a private key and the other is the corresponding public key.

For instance, let two peers, A and B, wishing to communicate in a secure fashion. A inserts a tuple to be read only by B and to be removed only by A. The operations are as follows.

$$\begin{aligned}\texttt{A: out}(&<\texttt{field}_1,\ldots,\texttt{field}_n,\texttt{P}_B,\texttt{P}_A>)\\ &\texttt{B: rd}(<*,\ldots,*,\texttt{S}_B,\#>)\\ &\texttt{A: in}(<*,\ldots,*,\#,\texttt{S}_A>)\end{aligned}$$

where $\texttt{P}_A/\texttt{S}_A$ and $\texttt{P}_B/\texttt{S}_B$ are the public/private keys of A and B, respectively. Remember that `#` means no key is specified and that `*` stands for wildcard.

[9]This definition of visibility ensures that entities cannot discover services which are published in groups they are not supposed to see. Indeed, the `getServices()` returns notably the group identifiers of the group in which services are published.

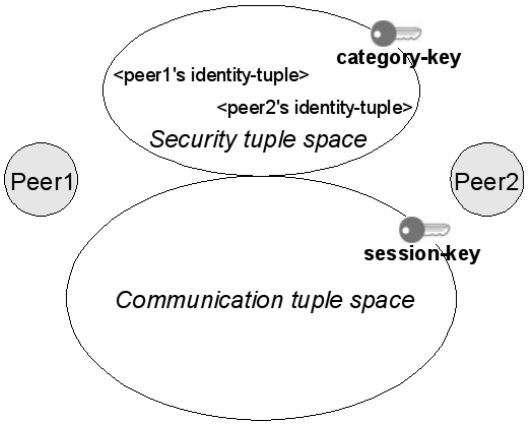


Figure 5.6: Group level design.

Design

The current design has to be reviewed to comply with *level 2*. At the tuple space level, the design is similar to the one presented in Section 5.1.1 (p. 109). An additional tuple space (called security tuple space) is used to manage access and key refresh of the discovery tuple spaces (*SD* and *GD*). Figure 5.7 (p. 122) illustrates them. The difference with the session-key design (presented in Section 5.1.1) is that this additional tuple space is not protected by symmetric keys. Instead, all communications are protected using the peers' key couples. Therefore, communications inside those tuple spaces are one-to-one.

Similarly, a security tuple space is added to each group tuple space. All communication tuple spaces are protected by session keys. The security tuple space is protected by the category key corresponding to the group (this ensures visibility restrictions). Furthermore, all communications are protected using the peers' key couple (this ensures access restrictions). Figure 5.6 illustrates the group level design. Optionally, service interactions could be protected using the invoker and provider's public keys. This ensures that invocation-tuples and reply-tuple can only be retrieved by the involved entities. Note that in this case, the provider must publish its public key in the *SD* tuple space (as an additional field of the service-tuple). Furthermore, the invoker must also provide its public key in the invocation-tuple.

The content of *GD* is similar to the previous design, but the reference-tuple contains an additional field specifying whether the group is static or not. Furthermore, if the group is dynamic, the reference-tuple contains the group public key. The group public key serves to ensure that a peer referencing a SMEPP dynamic group is a valid member. When a peer A wants to join a group, the member B handling A's request is challenged to prove B owns the private key corresponding to the public key it published in its reference-tuple.

The new level requires to review the way peers create and join groups. The solution we propose is as follows.

Group creation. To create a group, a peer A does the following steps:

1. The peer creates the security tuple space named by the group's identifier and protected by the category key corresponding to the wished visibility level.

2. It inserts a tuple ensuring its identity, called identity-tuple. An identity-tuple has the format: $<$`'_peer',peer_A,certificate_A,#,`$\mathtt{P}_A >$, where $\mathtt{P}_A$ is A's secret key.

3. It securely generates a fresh session key. If the group is dynamic, it also generates the group's asymmetric key pair.

4. It creates the communication tuple space protected by the session key and named by the group's identifier.

5. It registers a reaction on its local security tuple space. This reaction waits for a template <'_join',peer_X,certificate_X,challenge,S_A,S_A > (called join-tuple). When triggered, this reaction initiates the member approval process (see group joining). Furthermore, it registers a reaction for key refresh (see below).

6. It inserts a reference-tuple in *GD*. This tuple is read-protected by the group category key and remove-protected by P_A. If the group is dynamic, the tuple also contains the group public key.

Static group joining. In order to join a static group, a peer B does the following steps:

1. It creates the security tuple space corresponding to the group it wants to join. Remember this tuple space is protected by the category key and named by the group's identifier.

2. It invokes `rd()` operation on this tuple space to read an arbitrary identity-tuple. Let the owner of this tuple be peer A.

3. It checks that A's certificate is valid and that A has well the rights to have created this group.

4. It sends a join-tuple into A's local security tuple space:
<'_join',peer_id,certificate_B,challenge,P_A,P_A >.
The tuple contains B's identifier, B's certificate and a challenge generated by B. It is remove- and read-protected by A's public key. Thus, only A can retrieve it.

5. Through its registered reaction, A retrieves the join-tuple and checks that B's certificate allows it to join this group.

6. If B is allowed, A sends an authentication-tuple of the form:
<'_auth',{challenge}$_{S_A}$,sessionkey,P_B,P_B>.

7. B checks the validity of the challenge response. In case of failure, it aborts the joining process. This ensure that A is well the sender of the tuple.

8. B inserts its identity-tuple in the security tuple space.

9. B creates the communication tuple space corresponding to the group named by the group's identifier and protected by `sessionkey`.

10. B registers two reactions as in the 5^{th} step of group creation.

11. It finally inserts a reference-tuple corresponding to the group in *GD*.

Dynamic group joining. When a peer B wants to join a dynamic group, of which peer A is member, the following steps are necessary:

1. It creates the security tuple space corresponding to the group it wants to join (named by the group's identifier). Remember this tuple space is protected by the category key.

2. It invokes a `rd()` operation on this tuple space to read an arbitrary identity-tuple. Let the owner of this tuple be peer A.

3. It checks that A's certificate is valid and that A has the right to create groups of this visibility category.

4. B sends a join-tuple into A's local security tuple space.

5. Through its registered reaction, A retrieves the join-tuple. According to the group policy, A allows or not B to access the group. In a few words, classic policies could be[10]:

[10]This list is non-exhaustive. Note also that the implementation of policies is not discussed here as it would involve their definition in the SMEPP service model.

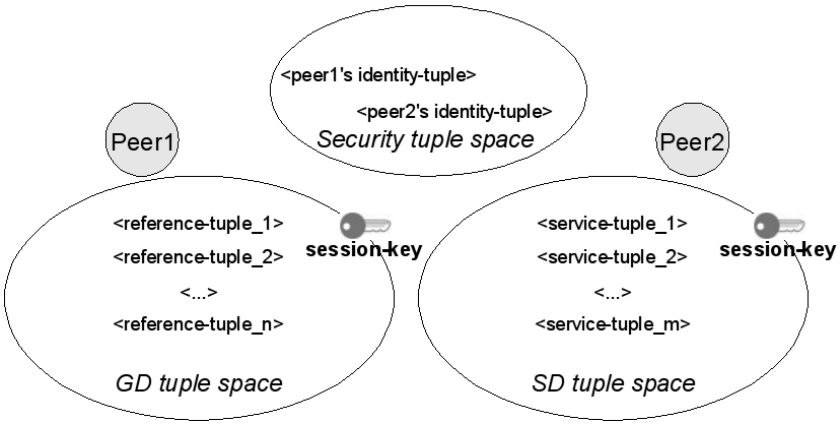


Figure 5.7: Application level design.

- *Open:* peers are always accepted,
- *Democracy:* there exists a mechanism inside the communication tuple space which allows the peers to vote. In this case, each of them must have an own vote policy.
- *Closed:* only a defined set of peers can enter the group. This implies that the members own a list of accepted peers.
- *Blacklist:* all peers are accepted except if they are blacklisted.

If B is accepted, A sends an authentication-tuple of the form:
$<$'_auth',$\{$challenge$\}_{S_A}$,$\{$challenge$\}_{S_G}$,sessionkey,asym_keys,P_B,$P_B>$.

6. B checks the validity of the challenge responses. In case of failure, it aborts the joining process. This ensures that A is well the tuple sender and that it is an authorised group member.

7. B inserts its identity-tuple in the security tuple space.

8. B creates the communication tuple space corresponding to the group. It is named by the group's identifier and protected by `sessionkey`.

9. B registers two reactions as in the 5^{th} step of group creation.

10. It finally inserts a reference-tuple corresponding to the group in *GD*. This tuple contains the group public key.

A mechanism of blacklisting could be simply modelled by a shared tuple space where every peer inserts tuples representing the peers it does not trust anymore. Thus, all peers could consult this tuple space before accepting a new member.

Application level security. At the application level, similar mechanisms are applied. Every peer starts by creating a security related tuple space. When a peer join the application and is the first to do it[11], it generates a session key to protect discovery tuple spaces. When other peers are already connected, a new peer joins the application by executing similar steps to group joining. Figure 5.7 illustrates the tuple spaces used to model the application level.

Refreshing keys. The key refresh mechanism is somewhat similar to the one described in Section 5.1.1 (p. 109). In order to refresh a group session key, a peer initiator inserts a refresh-tuple (similar to the one described in Section 5.1.1) in the security tuple space. However, in this case, the communication is one-to-one. Here, the initiator sends a tuple to each group member.

[11]In this case, no identity-tuples are available.

It contains the new session key and the current one. This last field ensures that the initiator is an actual member of the group. The tuple has thus the following form:
<'_refresh',initiator_id,current_sessionkey,new_sessionkey,P_X,P_X>.
This tuple has to be sent to every group member and it is important that the sending is done atomically. Indeed, if the initiator loses its connection while sending the tuples, two session keys would be used at the same time. To avoid this problem, the initiator has to use a *transactional* `outg()` operation as defined in the previous subsection. In this way, either all or no members receive the new key.

Every member has a reaction waiting for a refresh-tuple, similar to the one in Section 5.1.1. However, the two protection fields of the template contain the peer's secret key (S_x). When this reaction is triggered, the peer checks that the `current_sessionkey` is the key currently used to protect the communication tuple space and that the initiator's certificate is still valid. In this case, the peer changes the session key of the communication tuple space. Otherwise, the tuple is ignored.

Members exclusion. Members exclusion is a difficult problem. Indeed, it requires to specify a policy defining how to detect a member to exclude. It demands more complex security mechanisms and it also requires to prevent its misuse by malicious members. Note that, under our assumptions, members exclusion only applies to dynamic groups because static group access is based on static attributes contained in peers' certificates. Our goal here is not to define a complete design of a solution but rather to propose some leads that may be used to implement exclusion features.

The first topic is the definition of bad behaviours detection. In the design we proposed above, every peer could keep a list of the peers interacting with it. In this list, the peers could be ranked according to how well they behave. When a peer reaches a to-be-specified threshold of maliciousness, the peer holding the list can initiate an exclusion process. A simple way to assess a peer's trust rank could be to keep track of the security related interactions. For instance, when a peer notices another peer has failed a challenge or has an invalid certificate, this could arise suspicion on the peer's intention. The list owner could then decrease its trust rank.

To actually exclude a group member, a peer can initiate a session key refresh. Since the refresh related communication is one-to-one, it suffices that the initiator does not send the new key to the peer it wants to exclude. Furthermore, excluded peers could be blacklisted by others in order to prevent it to join again the group. This would require that the peers share blacklisting information.

It is important to note that allowing to exclude members could bring more problems than it could actually resolve. Indeed, if malicious peers have the possibility to exclude others, then they could exclude every suspicious member. To prevent this problem, a member exclusion should not be based on one's decision. This should involve a collaborative decision of group members, for instance, through a voting system.

5.3.2 Enhanced matching policy

In the proof-of-concept, groups and services discovery is implemented with the standard basic tuple matching policy provided by SecureLime. This implies that peers can only discover groups using their name and description in textual forms. In the same vein, the matching of service contracts is only based on a syntactical comparison of operation names and session types. For instance, consider a peer looking for a group called "plant-monitoring". In the current version of the implementation, if the peer enters a request containing the text "plantmonitoring", it will not get a positive result.

In addition, our implementation does not take into account the issues induced by embedded systems. Especially, in a real system, it would require mechanisms to reduce battery consumption and to overcome connectivity problems.

In the following, we propose different matching policies which may be helpful to improve the current groups and services discovery as well as embedded related features. However,

most of our propositions could be implemented at the application level (thus, on top of the coordination system). For instance, one can simulate finer grained matching policies on top of aggregate operations (i.e. `rdg()` and `ing()`). Still, we argue that providing these enhanced matching policies at the middleware level could be done in a more efficient way.

- *Range matching:* This matching policy permits to specify templates which feature values intervals. Tuples are matched if their values are in between the intervals. For instance, a template <`1-5`> would match the tuple <`3`>. A policy similar to range matching can be applied to unordered values by allowing several values in a template field. Fields match if the value in the tuple field is equal to one of the values in the template field. For instance, the template <`[water,tea,coffee],pie`> matches the tuple <`coffee,pie`>.

  In the context of SMEPP, this policy can be applied in several situations. Notably, it could be used to model Quality-of-Service (QoS) properties of services. An integer representing the level of QoS can be associated with each service. When a peer searches for services providing a certain level of QoS, it can specify the range including the QoS level it wants. Furthermore, the policy can also be used to discover groups. In particular, the policy can be used to search for group featuring a certain security or visibility level.

- *Comparative template:* Another interesting policy can allow to match the tuples comparatively. For instance, as for integer values, the policy selects a tuple with a field having the biggest, the least or the closest integer compared to the template. It may also compute distances between values. For instance, the policy could compute the distance between a string specified in the template and the strings contained in available tuples. Then, it would return the tuple with the closest field. In order to avoid irrelevant results when a distance is computed, it is necessary to be able to specify a bound to the wished distance.

  In SMEPP, this may be very useful to improve groups and services discovery. For instance, when a peer wants to discover groups related to printing, it specifies the key word "print" in the field corresponding to the group name. With this policy, the system returns the tuple(s) having a group name field similar to "print", for instance "printing" or "printer" etc.

- *Prioritised fields:* The fields can also be prioritised. One can imagine the user could associate an importance to each field in the template. The system then returns tuple(s) according to a certain computed score, based on the number of tuple matching the template and their priorities.

  In the context of SMEPP, this matching policy can be used in the services discovery. For instance, one could give a higher priority to the field representing the service name and a lower one to the QoS field.

- *Location based:* Some additional attributes can be associated with the tuples to add embedded related information about the host owning them, such as geographical coordinates, connection quality, host type (i.e. desktop, PDA, mobile phone, mote, etc.), battery autonomy etc. According to the type of attribute, the user could specify its needs in the template.

  For instance, in SMEPP, when a peer discovers services, the coordination system could first return the tuples owned by host having better battery autonomy or being the closest, etc. Likewise, the system could prioritise the tuples according to their host type (for instance, it could return the desktop tuples before the mote ones).

Obviously, if several policies are used at the same time, it is required to prioritise them. Such prioritisation should be made by the user. For instance, the coordination operations may require additional parameters specifying the order by which the policy have to be applied.

5.3.3 Interoperability

The current version of the implementation does not meet fully the SMEPP interoperability requirements. These requirements were left out of the thesis scope to focus on coordination

issues. The coordination system we chose does not provide many interoperability features. Indeed, SecureLime uses Java's serialised objects to exchange tuples between hosts. This implies that every device involved in the application network must be Java compliant. While Java is a ubiquitous language (it is available on many device types, e.g. from mobile phones to desktops), all devices are not Java compliant, especially the legacy ones. Thus, using SecureLime excludes this kind of hardware.

In order to develop an implementation meeting fully interoperability requirements on top of a coordination system, one needs a system which uses language and platform independent communications. For instance, the exchanged tuples could be in the form of structured ASCII files. Furthermore, a protocol shared by all entities is required to exchange the files between heterogeneous hosts (e.g. UDP/TCP protocols are available on almost all host types).

The Lime model can also be implemented in different languages. For instance, G.-P. Picco, in [39], proposes a coordination system based on Lime which is developed to run on embedded system. The implementation is written in nesC, a dialect of the C programming language and it is aimed to run on TinyOS, which is an operating system designed for wireless embedded sensor networks. This system, with additional security features, could be a good candidate to build an implementation meeting the interoperability requirements.

5.3.4 Implementation correctness

The SMEPP middleware intends to be used in various application fields. Some of them require strong reliability properties. Indeed, systems used in the context of plant monitoring, health care, disaster recovery, etc. must prohibit unexpected behaviour (e.g. software crashes, abnormal response time, erroneous behaviour, etc.). The purpose of the SMoL language is precisely to permit the theoretical analysis of SMEPP applications. However, the SMEPP middleware implementation and the translation from SMoL have to be trusted. On the one hand, the middleware needs to be proved reliable (i.e. compliant to the SMEPP service model). On the other hand, the translation needs to keep the semantics of SMoL programs.

Proving the middleware compliance amounts to analyse the primitives implementations with respect to the service model. This could be done by modelling the implementation using an abstract version of Java and SecureLime. In this way, it could be possible to compare this model with the SMEPP abstract model we presented in Section 2.3.3. Obviously, the modelling of the implementation using an abstract Java language is not a trivial work. However, the use of SecureLime could ease the comparison between the abstract model and the implementation model because SecureLime provides a higher abstraction level than basic coordination mechanisms (e.g. UDP/TCP protocols, RPC systems, etc.). Indeed, it is easier to compare a tuple space with a set and its operations than to compare the same set and operations with concepts such as TCP sockets and buffers.

To establish the equivalence between a SMoL program and its translation to Java, one also needs to build an abstract language which provides a higher level view of the implementation. Then, it is possible to analyse theoretically the correspondence of the implementation semantics and of the SMoL semantics. Thus, the purpose is that for each SMoL program P, its translation $T(P)$ keeps the same semantics. Since [15] defines a transformational semantics from SMoL to YAWL workflows, one could similarly model the implementation in YAWL concepts.

This proof scheme requires a deep understanding of SMoL and Java to be able to abstract them in a correct and sound way. Indeed, the rest of the proof assume the equivalence between the concrete language and its abstraction. Although such a proof would be of high importance in the final SMEPP implementation, we are forced to leave it as future work, given the amount of work it would require.

5.4 Concluding remarks

The previous sections elicited many features which have been showed to be very useful for service-oriented applications in ad hoc networks. In particular, we want to stress the lack of proper security mechanisms in coordination systems in general. SecSpaces [52] features by far

the most suitable security model in tuple space coordination language and its implementation, if it were peer-to-peer oriented, it would permit us to fully meet the highest SMEPP security requirements (i.e. to offer configurable security levels).

Chapter 6

Conclusion

The main motivation of this thesis is to study the development of a secure middleware for ad hoc systems on top of a coordination language. Ad hoc systems raise two main challenges to be faced. On the one hand, the devices involved in such systems lack of reliability. Indeed, these devices are prone to frequent and unpredicted disconnections and interact using unreliable communication technologies (i.e. wireless links). On the other hand, given the communication technologies typically used in ad hoc settings, the interactions need to be protected against eavesdropping, man-in-the-middle attack, denial-of-service, etc. Those two drawbacks are worsened by the absence of a centralised entity which, in other distributed models, takes care of monitoring the network topology and securing the communications.

These challenges must be addressed by a suitable middleware which wraps the difficulties induced by ad hoc networks. We showed that the development of this kind of middleware is eased by the separated specification of the internal behaviour of the devices and their interactions. Gelernter and Carriero, in [51], suggest to use a coordination model to specify the interaction part of distributed systems. Many coordination models have been proposed (e.g. PVM [103], RPC [9], Manifold [3], Gamma [5], etc.), but the advantages of the tuple space model made it become ubiquitous and widely used in the context of web-based applications [94]. This thesis experimented the development of a proof-of-concept of the SMEPP middleware, presented in Chapter 2, on top of a tuple based coordination system.

Firstly, we studied the state of the art of coordination languages to select the most suitable basis on top of which the middleware implementation is built. Chapter 3 gave a broad survey of the coordination theory in the light of the security and ad hoc requirements. We believe this analysis to be a good basis for the selection of an ad hoc oriented coordination system.

Then, we compared the SMEPP middleware requirements with the different features of the languages surveyed in the study. This led us to choose SecureLime as a basis for our implementation. Indeed, it provides features such as federated (password-protected) tuple spaces and tuple level access control. These features allowed us to successfully model all SMEPP key concepts.

We consider the experiment was successful as the implementation complies with the SMEPP objectives and requirements [96]. Furthermore, this work showed how a real world service model can be implemented using a tuple space coordination language. We argue the proof-of-concept design and implementation feature the following main contributions. Firstly, we provide an extension of SecureLime with the SMEPP programming model that offers a simple high-level API supporting the definition of peer and service code, peer groups, group-wise security, synchronous and asynchronous message patterns (one-to-one, direct or blind operation invocations) and event-based communication (one-to-many). Furthermore, this proof-of-concept can be used to test/simulate interactions of peers and services.

Part of Chapter 4 describing the middleware design and implementation was published at the *Workshop on Coordination Models and Applications: Knowledge in Pervasive Environments* (CoMA) held at *the 17th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises* (WETICE). The paper is available in Appendix C.

We have also implemented a SMoL to Java translator (which generates Java code from a SMoL specification) and we have integrated it with our implementation based on SecureLime. The resulting prototype produces executable Java code that runs on top of SecureLime starting from a SMoL description of the behaviour of a peer or service.

The critical analysis given in Chapter 5 of our implementation allowed us to elicit some features lacking in the currently available coordination systems. In particular, in order to enhance the security level (i.e. to build an asymmetric keys based system) addressed in the scope of our thesis, we proposed a set of requirements to be met by a coordination system. However, while theoretical models (partly) meet them, the state of the art analysis showed that no such coordination systems are available.

While the context of our work was restrictive, we believe the design ideas underlying our implementation and the new leads we propose in Chapter 5 can be re-used in similar projects. Indeed, the SMEPP project intends to produce a generic service-oriented middleware. Thus, concepts similar to the ones we implemented are likely to be used in other projects. Furthermore, there exist many coordination languages having common features with SecureLime. Therefore, it is possible to apply our high level design to other tuple space based languages.

Our goal was to study the development of a tuple space based secure middleware in ad hoc settings. We believe this goal has been achieved through the following contributions. We highlighted the needed features to be met by a coordination language for ad hoc networks, we provided a re-usable high level design for tuple space based application and we elicited requirements necessary to provide decentralised security management in tuple space systems.

However, in order to meet thoroughly the SMEPP requirements, we showed that the surveyed systems need additional features. In particular, these features must provide an interoperable communication model for embedded devices, advanced matching policies and asymmetric key based security mechanisms. We stressed the fact that even though some theoretical models exhibit these features, none of them is currently implemented. And yet, the lack of such implementations is manifest.

Bibliography

[1] Sudhir Ahuja, Nicholas J. Carriero, David H. Gelernter, and Venkatesh Krishnaswamy. Matching language and hardware for parallel computation in the linda machine. *IEEE Trans. Comput.*, 37(8):921–929, 1988.

[2] Amy L. Murphy, Gian Pietro Picco, Gruia-Catalin Roman and Radu Handorean. Lime and SecureLime Java Documentation.

[3] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

[4] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, 2004.

[5] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Commun. ACM*, 36(1):98–111, 1993.

[6] TU Berlin and University of Bologna. The pagespace project. `http://flp.cs.tu-berlin.de/pagespc/`. [Last date of access: July 2008].

[7] L. Bettini. X-klaim: a programming language for object oriented mobile code. user's manual. `http://music.dsi.unifi.it/download/xklaim.pdf`. [Last date of access: July 2008].

[8] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. Klava: a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.

[9] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 17(5):3, 1983.

[10] R. Bjornson. *Linda on distributed memory multiprocessors*. PhD thesis, Yale University. YALEU/DCS/RR-931, 1992.

[11] Gary Breed. Wireless ad hoc networks: Basic concepts. pages 44–46, March 2007.

[12] A. Brogi and R. Popescu. From bpel processes to yawl workflows. In M. Bravetti, M. Nunez, and G. Zavattaro, editors, *Proceedings of WS-FM'06, LNCS*, volume 4184, pages 107–122, 2006.

[13] A. Brogi, R. Popescu, F. Gutierrez, P. Lopez, and E. Pimentel. A service-oriented model for embedded peer-to-peer systems. 2007.

[14] Antonio Brogi and Razvan Popescu. Workflow semantics of peer and service behaviour. *TASE*, 2008.

[15] Antonio Brogi and Razvan Popescu. Workflow semantics of peer and service behaviour. *tase*, 0:143–150, 2008.

[16] Ciarán Bryce and Marco Cremonini. Coordination and security on the internet. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 274–298. Springer, 2001.

[17] Ciarán Bryce, Manuel Oriol, and Jan Vitek. A coordination model agents based on secure spaces. In *COORDINATION '99: Proceedings of the Third International Conference on Coordination Languages and Models*, pages 4–20, London, UK, 1999. Springer-Verlag.

[18] Ciarán Bryce and Jan Vitek. The javaseal mobile agent kernel. *Autonomous Agents and Multi-Agent Systems*, 4(4):359–384, 2001.

[19] Jenna Burrell, Tim Brooke, and Richard Beckwith. Vineyard computing: Sensor networks in agricultural production. *IEEE Pervasive Computing*, 3(1):38–45, 2004.

[20] Nadia Busi, Paolo Ciancarini, Roberto Gorrieri, and Gianluigi Zavattaro. Coordination models: A guided tour. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 6–24. Springer, 2001.

[21] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. Three semantics of the output operation for generative communication. In *COORDINATION '97: Proceedings of the Second International Conference on Coordination Languages and Models*, pages 205–219, London, UK, 1997. Springer-Verlag.

[22] Nadia Busi, Alberto Montresor, and Gianluigi Zavattaro. Data-driven coordination in peer-to-peer information systems. *International Journal of Cooperative Information Systems*, 13(1):63–89, March 2004.

[23] G. Cabri, L. Leonardi, G. Reggiani, and F. Zambonelli. Design and implementation of a programmable coordination architecture for mobile agents, 1999.

[24] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Mars: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[25] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Mobile-agent coordination models for internet applications. *Computer*, 33(2):82–89, 2000.

[26] Tracy Camp, Jeff Boleng, and Vanessa Davies. A survey of mobility models for ad hoc network research. *Wireless Communications and Mobile Computing (WCMC): Special issue on Mobile Ad Hoc Networking: Research, Trends and Applications*, 2:483–502, 2002.

[27] Nicholas Carriero and David Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Comput. Surv.*, 21(3):323–357, 1989.

[28] Nicholas Carriero and David Gelernter. Case studies in asynchronous data parallelism. *Int. J. Parallel Program.*, 22(2):129–149, 1994.

[29] Computer Security Center. *Trusted Computer Systems Evaluation Criteria.* Techincal Report CSC-STD-001-83, DoD Computer Security Center, Fort MEade, MD, 1983.

[30] V L Chung and C S McDonald. The development of a distributed capability system for vlos. *Aust. Comput. Sci. Commun.*, 24(3):57–64, 2002.

[31] P. Ciancarini, K. K. Jensen, and D. Yankelevich. On the operational semantics of a coordination language. In *Object-Based Models and Languages for Concurrent Systems*, volume 924, pages 77–106. Springer-Verlag New York, Inc., 1995.

[32] Paolo Ciancarini. Coordination models and languages as software integrators. *ACM Comput. Surv.*, 28(2):300–302, 1996.

[33] Paolo Ciancarini, Adreas Knoche, Robert Tolksdorf, and Fabio Vitali. Pagespace: an architecture to coordinate distributed applications on the web. In *Proceedings of the fifth international World Wide Web conference on Computer networks and ISDN systems*, pages 941–952, Amsterdam, The Netherlands, The Netherlands, 1996. Elsevier Science Publishers B. V.

[34] Paolo Ciancarini, Andrea Omicini, and Franco Zambonelli. Coordination technologies for internet agents. *Nordic J. of Computing*, 6(3):215–240, 1999.

[35] Paolo Ciancarini, Andrea Omicini, and Franco Zambonelli. Multiagent system engineering: The coordination viewpoint. In *ATAL '99: 6th International Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL),*, pages 250–259, London, UK, 2000. Springer-Verlag.

[36] Paolo Ciancarini and Davide Rossi. Jada - coordination and communication for java agents. In *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 213–226, London, UK, 1997. Springer-Verlag.

[37] Paolo Ciancarini, Robert Tolksdorf, and Fabio Vitali. The world wide web as a place for agents. In *Artificial Intelligence Today*, pages 175–193. Springer, 1999.

[38] Paolo Ciancarini, Robert Tolksdorf, Fabio Vitali, Davide Rossi, and Andreas Knoche. Coordinating multiagent applications on the www: A reference architecture. *IEEE Transactions on Software Engineering*, 24(5):362–375, 1998.

[39] Paolo Costa, Luca Mottola, Amy L. Murphy, and Gian Pietro Picco. Teenylime: transiently shared tuple space middleware for wireless sensor networks. In *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks*, pages 43–48, New York, NY, USA, 2006. ACM.

[40] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. Tinylime: Bridging mobile and sensor networks through middleware. *percom*, 00:61–72, 2005.

[41] Rocco de Nicola, Gian Luigi Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing)*, 1998.

[42] E. Denti, A. Omicini, and V. Toschi. Coordination technology for the development of multi-agent systems on the web, 1999.

[43] Enrico Denti, Antonio Natali, and Andrea Omicini. Programmable coordination media. In *COORDINATION '97: Proceedings of the Second International Conference on Coordination Languages and Models*, pages 274–288, London, UK, 1997. Springer-Verlag.

[44] Enrico Denti, Antonio Natali, and Andrea Omicini. On the expressive power of a language for programming coordination media. In *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing*, pages 169–177, New York, NY, USA, 1998. ACM.

[45] Enrico Denti and Andrea Omicini. An architecture for tuple-based coordination of multi-agent systems. *Softw. Pract. Exper.*, 29(12):1103–1121, 1999.

[46] Craig Faasen. Intermediate uniformly distributed tuple space on transputer meshes. In *Research Directions in High-Level Parallel Programming Languages*, pages 157–173, London, UK, 1992. Springer-Verlag.

[47] MPI Forum. Mpi specifications. `http://www.mpi-forum.org/`. [Last date of access: July 2008].

[48] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[49] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[50] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[51] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[52] Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. Supporting secure coordination in secspaces. *Fundam. Inf.*, 73(4):479–506, 2006.

[53] Object Management Group. Corba namingservice specification. `http://www.omg.org/cgi-bin/doc?formal/04-10-03`. [Last date of access: July 2008].

[54] Object Management Group. Corba specification. `http://www.omg.org/`. [Last date of access: July 2008].

[55] Radu Handorean and Gruia-Catalin Roman. Service provision in ad hoc networks. In *Coordination Models and Languages*, pages 207–219, 2002.

[56] Radu Handorean and Gruia-Catalin Roman. Secure sharing of tuple spaces in ad hoc settings. *ENTCS*, 85(3), 2003.

[57] Klaus Herrmann, Gero Mühl, and Michael A. Jaeger. Meshmdl event spaces - a coordination middleware for self-organizing applications in ad hoc networks. *Pervasive Mob. Comput.*, 3(4):467–487, 2007.

[58] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile environment. *Wirel. Netw.*, 10(6):643–652, 2004.

[59] IBM. Aglets. `http://www.trl.ibm.com/aglets/`. [Last date of access: July 2008].

[60] Jean-Marie Jacquet and Isabelle Linden. Coordinating context-aware applications in modbile ad hoc networks. In *First ERCIM workshop on eMobility*, 2007.

[61] JXTA. `http://www.jxta.org/`. [Last date of access: July 2008].

[62] V. Krishnaswamy, S. Ahuja, N. Carriero, and D. Gelernter. The architecture of a linda coprocessor. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 240–249, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[63] Konrad Lorincz, David J. Malan, Thaddeus R. F. Fulford-Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoffrey Mainland, Matt Welsh, and Steve Moulton. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing*, 3(4):16–23, 2004.

[64] Microsoft. Dcom specification. `http://www.microsoft.com/`. [Last date of access: July 2008].

[65] Sun Microsystems. Java specifications. `http://java.sun.com/`. [Last date of access: July 2008].

[66] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[67] Robin Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[68] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. Technical report, 2002.

[69] N. H. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 322, Washington, DC, USA, 1998. IEEE Computer Society.

[70] Naftaly H. Minsky. The imposition of protocols over open distributed systems. *IEEE Trans. Softw. Eng.*, 17(2):183–195, 1991.

[71] Naftaly H. Minsky and Jerrold Leichter. Law-governed linda as a coordination model. In *ECOOP '94: Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, pages 125–146, London, UK, 1995. Springer-Verlag.

[72] Naftaly H. Minsky, Yaron M. Minsky, and Victoria Ungureanu. Making tuple spaces safe for heterogeneous distributed systems. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 218–226, New York, NY, USA, 2000. ACM.

[73] Naftaly H. Minsky and Victoria Ungureanu. Unified support for heterogeneous security policies in distributed systems. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium, 1998*, pages 10–10, Berkeley, CA, USA, 1998. USENIX Association.

[74] Natfaly H. Minsky and Constantin Serban. Law governed interaction (lgi): A distributed coordination and control mechanism (an introduction, and a reference manual). `http://www.moses.rutgers.edu/manual.pdf`. [Last date of access: July 2008].

[75] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15(3):279–328, 2006.

[76] Bruce J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, may 2000.

[77] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Coordinating mobile agents via blackboards and access rights. In *COORDINATION '97: Proceedings of the Second International Conference on Coordination Languages and Models*, pages 220–237, London, UK, 1997. Springer-Verlag.

[78] OASIS. BPEL V2.0 Primer. `http://www.oasis-open.org/committees/download.php/23974/wsbpel-v2.0-primer.pdf`. [Last date of access: July 2008].

[79] OASIS. BPEL V2.0 Specification. `http://www.oasis-open.org/committees/download.php/23974/wsbpel-v2.0-primer.pdf`. [Last date of access: July 2008].

[80] University of Bologna. Soma. `http://lia.deis.unibo.it/Research/SOMA/`. [Last date of access: July 2008].

[81] Andrea Omicini. On the semantics of tuple-based coordination models. In *SAC '99: Proceedings of the 1999 ACM symposium on Applied computing*, pages 175–182, New York, NY, USA, 1999. ACM.

[82] Andrea Omicini. Coordination models and languages: State of the art, introduction. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 3–5. Springer, 2001.

[83] Andrea Omicini. Soda: societies and infrastructures in the analysis and design of agent-based systems. In *First international workshop, AOSE 2000 on Agent-oriented software engineering*, pages 185–193, Secaucus, NJ, USA, 2001. Springer-Verlag New York, Inc.

[84] Andrea Omicini and Franco Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999.

[85] Andrea Omicini, Franco Zambonelli, Matthias Klusch, and Robert Tolksdorf, editors. *Coordination of Internet Agents: Models, Technologies, and Applications*. Springer, 2001.

[86] George A. Papadopoulos. Models and technologies for the coordination of internet agents: A survey. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 25–56. Springer, 2001.

[87] Jose Luis Gonzalez Pierre Plaza and Julio Marina. From a client-server based e-health platform to a pervasive solution. 2007.

[88] LIME Project. Lime download page. `http://lime.sourceforge.net/downloads/index.html`. [Last date of access: July 2008].

[89] POPEYE Project. Peer to peer collaboration working environments over mobile ad-hoc networks, June 2008. `http://www.ist-popeye.eu/`. [Last date of access: July 2008].

[90] WORKPAD EU Project. An adaptive peer-to-peer software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios. `http://www.workpad-project.eu/home.jsp`. [Last date of access: June 2008].

[91] Gruia-Catalin Roman, Amy L. Murphy, and Gian Pietro Picco. Coordination and mobility. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 253–273. Springer, 2001.

[92] Davide Rossi. Jada specification. `http://www.cs.unibo.it/ rossi/jada/`. [Last date of access: July 2008].

[93] Davide Rossi, Giacomo Cabri, and Enrico Denti. Tuple-based technologies for coordination. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 83–109. Springer, 2001.

[94] Antony I. T. Rowstron. Run-time systems for coordination. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 61–82. Springer, 2001.

[95] Jonathan Strauss Shapiro. *Eros: a capability system*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 1999. Supervisor-David J. Farber and Supervisor-Jonathan M. Smith.

[96] SMEPP Coalition. D1.1: State of the art and generic middleware requirements. `http://www.smepp.org/`. [Last date of access: July 2008].

[97] SMEPP Coalition. D1.3: Application requirements. `http://www.smepp.org/`. newblock [Last date of access: July 2008].

[98] SMEPP Coalition. D2.1: Service model description. `http://www.smepp.org/`. newblock [Last date of access: July 2008].

[99] SMEPP Coalition. SMEPP Description of work.

[100] Sun. Using Java Reflection, Technical Article. `http://java.sun.com/developer/technicalArticles/ALT/Reflection/index.html`. newblock [Last date of access: June 2008].

[101] Sun. Java cryptography architecture (jca) reference guide. `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`. newblock [Last date of access: July 2008].

[102] Neelakantan Sundaresan and Vinay Rajagopalan. Java paradigms for mobile agent facilities. In *OOPSLA '97: Addendum to the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 133–135, New York, NY, USA, 1997. ACM.

[103] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The pvm concurrent computing system: evolution, experiences, and trends. *Parallel Comput.*, 20(4):531–545, 1994.

[104] Robert Tolksdorf. A machine for uncoupled coordination and its concurrent behavior. In *ECOOP '94: Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, pages 176–193, London, UK, 1995. Springer-Verlag.

[105] Robert Tolksdorf. Coordinating services in open distributed systems with laura. In *COORDINATION '96: Proceedings of the First International Conference on Coordination Languages and Models*, pages 386–402, London, UK, 1996. Springer-Verlag.

[106] Nur Izura Udzir, Alan M. Wood, and Jeremy L. Jacob. Coordination with multicapabilities. *Sci. Comput. Program.*, 64(2):205–222, 2007.

[107] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.

[108] W3C. Owl. http://www.w3.org/TR/owl-features. newblock [Last date of access: April 2008].

[109] W3C. Wsdl v1.1. http://www.w3.org/TR/wsdl. newblock [Last date of access: April 2008].

[110] W3C. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath newblock [Last date of access: April 2008].

[111] Jim Waldo. The jini architecture for network-centric computing. *Commun. ACM*, 42(7):76–82, 1999.

[112] A. Wood. Coordination with Attributes. In P. Ciancarini and A. Wolf, editors, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, volume 1594, pages 21–36, Amsterdam, Netherland, 1999. Springer-Verlag, Berlin.

[113] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T spaces. *IBM Syst. J.*, 37(3):454–474, 1998.

[114] YAWL. YAWL Web Site. http://www.yawlfoundation.org. newblock [Last date of access: June 2008].

[115] Gianluigi Zavattaro. *Coordination Models and Languages: Semantics and Expressiveness*. PhD thesis, Department of Computer Science, University of Bologna, Italy, 2000.

Appendix A

XML schemas

A.1 Service schemas

A.1.1 Signature.xsd schema

```
<?xml version="1.0" encoding="utf-8"?>
<!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by BlackJack (BlackJack) -->
<!--
    W3C XML Schema defined in the Web Services Description (WSDL)
     Version 2.0 specification
      http://www.w3.org/TR/wsdl20

    Copyright  2005 World Wide Web Consortium,

    (Massachusetts Institute of Technology, European Research Consortium for
    Informatics and Mathematics, Keio University). All Rights Reserved. This
    work is distributed under the W3C Software License [1] in the hope that
    it will be useful, but WITHOUT ANY WARRANTY; without even the implied
    warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

    [1] http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231

    $Id: wsdl20.xsd,v 1.1 2007/05/17 16:13:03 plehegar Exp $
-->
<xs:schema targetNamespace="http://www.w3.org/ns/wsdl" xmlns:sign="http://www.w3.org/ns/wsdl"
      xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
      attributeFormDefault="unqualified">
         <xs:element name="documentation" type="sign:DocumentationType"/>
         <xs:complexType name="DocumentationType" mixed="true">
                  <xs:sequence>
                           <xs:any namespace="##any" processContents="lax" minOccurs="0"
                                    maxOccurs="unbounded"/>
                  </xs:sequence>
                  <xs:anyAttribute namespace="##other" processContents="lax"/>
         </xs:complexType>
         <xs:complexType name="DocumentedType" mixed="false">
                  <xs:annotation>
                           <xs:documentation>
       This type is extended by component types to allow them to be documented.
                           </xs:documentation>
                  </xs:annotation>
                  <xs:sequence>
                           <xs:element ref="sign:documentation" minOccurs="0" maxOccurs="unbounded"/>
                  </xs:sequence>
         </xs:complexType>
         <xs:complexType name="ExtensibleDocumentedType" abstract="true" mixed="false">
                  <xs:annotation>
                           <xs:documentation>
       This type is extended by component types to allow
       attributes from other namespaces to be added.
                           </xs:documentation>
```

```
            </xs:annotation>
            <xs:complexContent mixed="false">
                <xs:extension base="sign:DocumentedType">
                    <xs:anyAttribute processContents="lax"/>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
51      <xs:complexType name="DescriptionType" mixed="false">
            <xs:annotation>
                <xs:documentation>
        Although correct, this type declaration does not capture
        all the constraints on the contents of the wsdl:description
        element as defined by the WSDL 2.0 specification.

        In particular, the ordering constraints wrt elements preceding
        and following the wsdl:types child element are not captured, as
        attempts to incorporate such restrictions in the schema
61      ran afoul of the UPA (Unique Particle Attribution) rule
        in the XML Schema language.

        Please refer to the WSDL 2.0 specification for
        additional information on the contents of this type.
                </xs:documentation>
            </xs:annotation>
            <xs:complexContent mixed="false">
                <xs:extension base="sign:ExtensibleDocumentedType">
                    <xs:sequence>
71                      <xs:element ref="sign:types" minOccurs="0"
                            maxOccurs="unbounded"/>
                        <xs:element ref="sign:interface"/>
                        <xs:any namespace="##other" processContents="lax"
                            minOccurs="0"/>
                    </xs:sequence>
                    <xs:attribute name="targetNamespace" type="xs:anyURI"
                        use="optional"/>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
        <!-- types for import and include elements -->
        <xs:element name="types" type="sign:TypesType"/>
81      <xs:complexType name="TypesType" mixed="false">
            <xs:complexContent mixed="false">
                <xs:extension base="sign:ExtensibleDocumentedType">
                    <xs:sequence>
                        <xs:any namespace="##other" processContents="strict"
                            minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
        <!-- parts related to wsdl:interface -->
91      <xs:element name="interface" type="sign:InterfaceType">
            <xs:unique name="operation">
                <xs:selector xpath="sign:operation"/>
                <xs:field xpath="@name"/>
            </xs:unique>
            <xs:unique name="fault">
                <xs:selector xpath="sign:fault"/>
                <xs:field xpath="@name"/>
            </xs:unique>
        </xs:element>
101     <xs:complexType name="InterfaceType" mixed="false">
            <xs:complexContent mixed="false">
                <xs:extension base="sign:ExtensibleDocumentedType">
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="operation"
                            type="sign:InterfaceOperationType"/>
                        <xs:element name="fault" type="sign:InterfaceFaultType"/>
                        <xs:any namespace="##other" processContents="lax"/>
                    </xs:choice>
```

```
                        <xs:attribute name="name" type="xs:NCName" use="required"/>
                        <xs:attribute name="extends" use="optional">
111                         <xs:simpleType>
                                <xs:list itemType="xs:QName"/>
                            </xs:simpleType>
                        </xs:attribute>
                        <xs:attribute name="styleDefault" use="optional">
                            <xs:simpleType>
                                <xs:list itemType="xs:anyURI"/>
                            </xs:simpleType>
                        </xs:attribute>
                    </xs:extension>
121             </xs:complexContent>
        </xs:complexType>
        <xs:complexType name="InterfaceOperationType" mixed="false">
            <xs:complexContent mixed="false">
                <xs:extension base="sign:ExtensibleDocumentedType">
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="input" type="sign:MessageRefType"/>
                        <xs:element name="output" type="sign:MessageRefType"/>
                        <xs:element name="infault" type="sign:MessageRefFaultType"/>
                        <xs:element name="outfault" type="sign:MessageRefFaultType"/>
131                     <xs:any namespace="##other" processContents="lax"/>
                    </xs:choice>
                    <xs:attribute name="name" type="xs:NCName" use="required"/>
                    <xs:attribute name="pattern" type="xs:anyURI" use="optional"/>
                    <xs:attribute name="safe" type="xs:boolean" use="optional"/>
                    <xs:attribute name="style" type="xs:anyURI" use="optional"/>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
        <xs:complexType name="MessageRefType" mixed="false">
141         <xs:complexContent mixed="false">
                <xs:extension base="sign:ExtensibleDocumentedType">
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:any namespace="##other" processContents="lax"/>
                    </xs:choice>
                    <xs:attribute name="messageLabel" type="xs:NCName" use="optional"/>
                    <xs:attribute name="element" type="sign:ElementReferenceType"
                        use="optional"/>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
151     <xs:simpleType name="ElementReferenceType">
            <xs:annotation>
                <xs:documentation>
Use the QName of a GED that describes the content,
#any for any content,
#none for empty content, or
#other for content described by some other extension attribute that references a declaration in a
    non-XML extension type system.
</xs:documentation>
            </xs:annotation>
            <xs:union memberTypes="xs:QName">
161             <xs:simpleType>
                    <xs:restriction base="xs:token">
                        <xs:enumeration value="#any"/>
                        <xs:enumeration value="#none"/>
                        <xs:enumeration value="#other"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:union>
        </xs:simpleType>
        <xs:complexType name="MessageRefFaultType" mixed="false">
171         <xs:complexContent mixed="false">
                <xs:extension base="sign:ExtensibleDocumentedType">
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:any namespace="##other" processContents="lax"/>
                    </xs:choice>
                    <xs:attribute name="ref" type="xs:QName" use="required"/>
```

```
                        <xs:attribute name="messageLabel" type="xs:NCName" use="optional"/>
                    </xs:extension>
                </xs:complexContent>
            </xs:complexType>
181         <xs:complexType name="InterfaceFaultType" mixed="false">
                <xs:complexContent mixed="false">
                    <xs:extension base="sign:ExtensibleDocumentedType">
                        <xs:choice minOccurs="0" maxOccurs="unbounded">
                            <xs:any namespace="##other" processContents="lax"/>
                        </xs:choice>
                        <xs:attribute name="name" type="xs:NCName" use="required"/>
                        <xs:attribute name="element" type="xs:QName" use="optional"/>
                    </xs:extension>
                </xs:complexContent>
191         </xs:complexType>
            <!-- types related to wsdl:binding -->
            <!-- types related to service -->
            <xs:attribute name="required" type="xs:boolean"/>
            <xs:complexType name="ExtensionElement" abstract="true" mixed="false">
                <xs:annotation>
                    <xs:documentation>
              This abstract type is intended to serve as the base type for
          extension elements. It includes the wsdl:required attribute
          which it is anticipated will be used by most extension elements
201           </xs:documentation>
                </xs:annotation>
                <xs:attribute ref="sign:required" use="optional"/>
            </xs:complexType>
            <xs:element name="description" type="sign:DescriptionType"/>
    </xs:schema>
```

A.1.2 Contract.xsd schema

```
    <?xml version="1.0" encoding="UTF-8"?>
    <!-- edited with XMLSPY v5 rel. 4 U (http://www.xmlspy.com) by BlackJack (BlackJack) -->
  3 <xs:schema xmlns:sign="http://www.w3.org/ns/wsdl" xmlns:smol="http://SMoL.SMEPP.org"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
        attributeFormDefault="unqualified" blockDefault="#all">
        <xs:import namespace="http://www.w3.org/ns/wsdl" schemaLocation="Signature.xsd"/>
        <xs:import namespace="http://SMoL.SMEPP.org" schemaLocation="SMoLv21.xsd"/>
        <xs:element name="Contract">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="Signature">
            <xs:complexType>
             <xs:sequence>
              <xs:element name="service">
 13            <xs:complexType>
               <xs:simpleContent>
                <xs:extension base="xs:string">
                 <xs:attribute name="name" type="xs:QName" use="required"/>
                 <xs:attribute name="category" type="xs:QName" use="required"/>
                 <xs:attribute name="type" use="optional" default="state-less">
                  <xs:simpleType>
                   <xs:restriction base="xs:NMTOKEN">
                    <xs:enumeration value="state-less"/>
                    <xs:enumeration value="state-full"/>
 23                 </xs:restriction>
                  </xs:simpleType>
                 </xs:attribute>
                 <xs:attribute name="sessionState" use="optional" default="session-less">
                  <xs:simpleType>
                   <xs:restriction base="xs:NMTOKEN">
                    <xs:enumeration value="session-less"/>
                    <xs:enumeration value="session-full"/>
                   </xs:restriction>
                  </xs:simpleType>
```

```
                </xs:attribute>
               </xs:extension>
              </xs:simpleContent>
             </xs:complexType>
            </xs:element>
            <xs:element ref="sign:description"/>
           </xs:sequence>
           <xs:attributeGroup ref="xs:ontologyAttrG"/>
          </xs:complexType>
         </xs:element>
         <xs:element name="Grounding">
          <xs:complexType>
           <xs:sequence>
            <xs:element name="Parameters" minOccurs="0">
             <xs:complexType>
              <xs:sequence>
               <xs:element name="param" maxOccurs="unbounded">
                <xs:complexType>
                 <xs:simpleContent>
                  <xs:extension base="xs:string">
                   <xs:attribute name="name" type="xs:QName" use="required"/>
                   <xs:attribute name="type" type="xs:QName" use="required"/>
                  </xs:extension>
                 </xs:simpleContent>
                </xs:complexType>
               </xs:element>
              </xs:sequence>
              <xs:attributeGroup ref="xs:ontologyAttrG"/>
             </xs:complexType>
            </xs:element>
            <xs:any namespace="##other" minOccurs="0"/>
           </xs:sequence>
          </xs:complexType>
         </xs:element>
         <xs:element name="Properties" minOccurs="0">
          <xs:complexType>
           <xs:sequence>
            <xs:element name="property" maxOccurs="unbounded">
             <xs:complexType>
              <xs:simpleContent>
               <xs:extension base="xs:string">
                <xs:attribute name="category" type="xs:QName" use="required"/>
                <xs:attribute name="name" type="xs:QName" use="required"/>
                <xs:attribute name="type" type="xs:QName" use="required"/>
                <xs:anyAttribute/>
               </xs:extension>
              </xs:simpleContent>
             </xs:complexType>
            </xs:element>
           </xs:sequence>
           <xs:attributeGroup ref="xs:ontologyAttrG"/>
          </xs:complexType>
         </xs:element>
         <xs:element name="QoS" minOccurs="0">
          <xs:complexType>
           <xs:sequence>
            <xs:element name="QoSParameter" maxOccurs="unbounded">
             <xs:complexType>
              <xs:sequence>
               <xs:element name="QoSImpact" type="xs:string" minOccurs="0"
                    maxOccurs="unbounded"/>
               <xs:element name="QoSMetric" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                 <xs:simpleContent>
                  <xs:extension base="xs:string">
                   <xs:attribute name="type" type="xs:QName" use="required"/>
                   <xs:attribute name="unit" type="xs:QName" use="optional"/>
                  </xs:extension>
                 </xs:simpleContent>
                </xs:complexType>
```

```
                    </xs:element>
103                 <xs:element name="RelationShip" minOccurs="0">
                     <xs:complexType>
                      <xs:sequence>
                       <xs:element name="Relation" maxOccurs="unbounded">
                        <xs:complexType>
                         <xs:attribute name="refQoSParameter" type="xs:string" use="required"/>
                         <xs:attribute name="impactFactor" use="required">
                          <xs:simpleType>
                           <xs:restriction base="xs:QName">
                            <xs:enumeration value="Proportional"/>
113                         <xs:enumeration value="InverselyProportional"/>
                           </xs:restriction>
                          </xs:simpleType>
                         </xs:attribute>
                        </xs:complexType>
                       </xs:element>
                      </xs:sequence>
                     </xs:complexType>
                    </xs:element>
                    <xs:element name="Agregations" minOccurs="0">
123                  <xs:complexType>
                      <xs:sequence>
                       <xs:element name="Aggregated" maxOccurs="unbounded">
                        <xs:complexType>
                         <xs:attribute name="refQoSParameter" type="xs:string" use="required"/>
                        </xs:complexType>
                       </xs:element>
                      </xs:sequence>
                     </xs:complexType>
                    </xs:element>
133                 </xs:sequence>
                    <xs:attribute name="nature" use="required">
                     <xs:simpleType>
                      <xs:restriction base="xs:NMTOKEN">
                       <xs:enumeration value="static"/>
                       <xs:enumeration value="dynamic"/>
                      </xs:restriction>
                     </xs:simpleType>
                    </xs:attribute>
                    <xs:attribute name="name" type="xs:QName" use="required"/>
143                 <xs:attribute name="domain" type="xs:QName" use="required"/>
                   </xs:complexType>
                  </xs:element>
                 </xs:sequence>
                 <xs:attributeGroup ref="xs:ontologyAttrG"/>
                </xs:complexType>
               </xs:element>
               <xs:element name="Behavior" minOccurs="0">
                <xs:complexType>
                 <xs:sequence>
153               <xs:element ref="smol:process"/>
                 </xs:sequence>
                </xs:complexType>
               </xs:element>
              </xs:sequence>
             </xs:complexType>
            </xs:element>
            <xs:attributeGroup name="ontologyAttrG">
                    <xs:attribute name="targetTaxNamespace" type="xs:anyURI" use="optional"/>
                    <xs:attribute name="targetOwnNamespace" type="xs:anyURI" use="optional"/>
163         </xs:attributeGroup>
</xs:schema>
```

A.2 SMoL schema

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!--
        Copyright (c) OASIS Open 2003-2007. All Rights Reserved.
        Modified by Razvan Popescu (UPI) for SMEPP.
-->
<xsd:schema xmlns="http://SMoL.SMEPP.org" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://SMoL.SMEPP.org" elementFormDefault="qualified" blockDefault="#all">
        <xsd:import namespace="http://www.w3.org/XML/1998/namespace"
              schemaLocation="http://www.w3.org/2001/xml.xsd"/>
        <xsd:element name="process" type="tProcess"/>
        <xsd:complexType name="tProcess">
                <xsd:complexContent>
                        <xsd:extension base="tExtensibleElements">
                                <xsd:sequence>
                                        <xsd:element ref="import" minOccurs="0"
                                              maxOccurs="unbounded"/>
                                        <xsd:element ref="variables" minOccurs="0"/>
                                        <xsd:group ref="activity"/>
                                </xsd:sequence>
                                <xsd:attribute name="name" type="xsd:NCName" use="required"/>
                                <xsd:attribute name="targetNamespace" type="xsd:anyURI"
                                      use="required"/>
                                <xsd:attribute name="queryLanguage" type="xsd:anyURI"
                                      default="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"/>
                                <xsd:attribute name="expressionLanguage" type="xsd:anyURI"
                                      default="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"/>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="tExtensibleElements">
                <xsd:annotation>
                        <xsd:documentation>
                                This type is extended by other component types to allow elements and
                                      attributes from
                                other namespaces to be added at the modeled places.
                        </xsd:documentation>
                </xsd:annotation>
                <xsd:sequence>
                        <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
                        <xsd:any namespace="##other" processContents="lax" minOccurs="0"
                              maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:anyAttribute namespace="##other" processContents="lax"/>
        </xsd:complexType>
        <xsd:element name="documentation" type="tDocumentation"/>
        <xsd:complexType name="tDocumentation" mixed="true">
                <xsd:sequence>
                        <xsd:any namespace="##any" processContents="lax" minOccurs="0"
                              maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="source" type="xsd:anyURI"/>
                <!--<xsd:attribute ref="xml:lang"/>-->
        </xsd:complexType>
        <xsd:element name="import" type="tImport"/>
        <xsd:complexType name="tImport">
                <xsd:complexContent>
                        <xsd:extension base="tExtensibleElements">
                                <xsd:attribute name="namespace" type="xsd:anyURI" use="optional"/>
                                <xsd:attribute name="location" type="xsd:anyURI" use="optional"/>
                                <xsd:attribute name="importType" type="xsd:anyURI" use="required"/>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="variables" type="tVariables"/>
        <xsd:complexType name="tVariables">
                <xsd:complexContent>
                        <xsd:extension base="tExtensibleElements">
                                <xsd:sequence>
                                        <xsd:element ref="variable" maxOccurs="unbounded"/>
                                </xsd:sequence>
                        </xsd:extension>
```

```
            </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="variable" type="tVariable"/>
        <xsd:complexType name="tVariable">
            <xsd:annotation>
                <xsd:documentation>
                    The "from" tag serves to initialise variables.
                </xsd:documentation>
            </xsd:annotation>
            <xsd:complexContent>
                <xsd:extension base="tExtensibleElements">
                    <xsd:sequence>
                        <xsd:element ref="from" minOccurs="0"/>
                    </xsd:sequence>
                    <xsd:attribute name="name" type="VariableName" use="required"/>
                    <xsd:attribute name="messageType" type="xsd:QName" use="optional"/>
                    <xsd:attribute name="type" type="xsd:QName" use="optional"/>
                    <xsd:attribute name="element" type="xsd:QName" use="optional"/>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
        <xsd:simpleType name="VariableName">
            <xsd:restriction base="xsd:NCName">
                <xsd:pattern value="[^\.]+"/>
            </xsd:restriction>
        </xsd:simpleType>
        <xsd:group name="activity">
            <xsd:choice>
                <xsd:group ref="basic_commands"/>
                <xsd:group ref="structured_commands"/>
            </xsd:choice>
        </xsd:group>
        <xsd:group name="basic_commands">
            <xsd:choice>
                <xsd:group ref="primitives"/>
                <xsd:element ref="empty"/>
                <xsd:element ref="assign"/>
                <xsd:element ref="wait"/>
                <xsd:element ref="throw"/>
                <xsd:element ref="catch"/>
                <!-- to be used only inside faultHandlers -->
                <xsd:element ref="catchAll"/>
                <!-- to be used only inside faultHandlers -->
                <xsd:element ref="exit"/>
            </xsd:choice>
        </xsd:group>
        <xsd:group name="structured_commands">
            <xsd:choice>
                <xsd:element ref="sequence"/>
                <xsd:element ref="flow"/>
                <xsd:element ref="while"/>
                <xsd:element ref="repeatUntil"/>
                <xsd:element ref="if"/>
                <xsd:element ref="pick"/>
                <xsd:element ref="informationHandler"/>
                <xsd:element ref="faultHandler"/>
            </xsd:choice>
        </xsd:group>
        <xsd:group name="primitives">
            <xsd:choice>
                <!-- Peer Management Primitives -->
                <xsd:element ref="newPeer"/>
                <xsd:element ref="getPeerId"/>
                <xsd:element ref="getPeers"/>
                <!-- Group Management Primitives -->
                <xsd:element ref="createGroup"/>
                <xsd:element ref="getGroups"/>
                <xsd:element ref="getGroupDescription"/>
                <xsd:element ref="joinGroup"/>
                <xsd:element ref="leaveGroup"/>
```

```
                        <xsd:element ref="getIncludingGroups"/>
                        <xsd:element ref="getPublishingGroup"/>
                        <!-- Service Management Primitives -->
                        <xsd:element ref="publish"/>
                        <xsd:element ref="unpublish"/>
                        <xsd:element ref="getServices"/>
                        <xsd:element ref="getServiceContract"/>
                        <xsd:element ref="startSession"/>
                        <!-- Message Management Primitives -->
                        <xsd:element ref="invoke"/>
                        <xsd:element ref="receiveMessage"/>
                        <xsd:element ref="reply"/>
                        <!-- Event Management Primitives -->
                        <xsd:element ref="event"/>
                        <xsd:element ref="receiveEvent"/>
                        <xsd:element ref="subscribe"/>
                        <xsd:element ref="unsubscribe"/>
                </xsd:choice>
        </xsd:group>
        <!-- Start of basic_commands' definition. -->
        <!-- Start of primitives' definition. -->
        <!-- Peer Management Primitives -->
        <!-- peerId newPeer(credentials) -->
        <xsd:element name="newPeer" type="tNewPeer"/>
        <xsd:complexType name="tNewPeer">
                <xsd:complexContent>
                        <xsd:extension base="tActivity">
                                <xsd:sequence>
                                        <xsd:element name="input" type="tInputNewPeer"/>
                                        <xsd:element name="output" type="tOutputNewPeer"
                                            minOccurs="0"/>
                                </xsd:sequence>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="tInputNewPeer">
                <xsd:sequence>
                        <xsd:element name="credentials" type="tFrom"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="tOutputNewPeer">
                <xsd:sequence>
                        <xsd:element name="peerId" type="tTo"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="tActivity">
                <xsd:complexContent>
                        <xsd:extension base="tExtensibleElements">
                                <xsd:attribute name="name" type="xsd:NCName"/>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <!-- peerId getPeerId(id?) -->
        <xsd:element name="getPeerId" type="tGetPeerId"/>
        <xsd:complexType name="tGetPeerId">
                <xsd:complexContent>
                        <xsd:extension base="tActivity">
                                <xsd:sequence>
                                        <xsd:element name="input" type="tInputGetPeerId"
                                            minOccurs="0"/>
                                        <xsd:element name="output" type="tOutputGetPeerId"
                                            minOccurs="0"/>
                                </xsd:sequence>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="tInputGetPeerId">
                <xsd:sequence>
                        <xsd:element name="id" type="tFrom"/>
                </xsd:sequence>
```

```
</xsd:complexType>
<xsd:complexType name="tOutputGetPeerId">
    <xsd:sequence>
        <xsd:element name="peerId" type="tTo"/>
    </xsd:sequence>
</xsd:complexType>
<!-- peerId[] getPeers(groupId?) -->
<xsd:element name="getPeers" type="tGetPeers"/>
<xsd:complexType name="tGetPeers">
    <xsd:complexContent>
        <xsd:extension base="tActivity">
            <xsd:sequence>
                <xsd:element name="input" type="tInputGetPeers"
                    minOccurs="0"/>
                <xsd:element name="output" type="tOutputGetPeers"
                    minOccurs="0"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputGetPeers">
    <xsd:sequence>
        <xsd:element name="groupId" type="tFrom"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputGetPeers">
    <xsd:sequence>
        <xsd:element name="peerIdArray" type="tTo"/>
    </xsd:sequence>
</xsd:complexType>
<!-- Group Management Primitives -->
<!-- groupId createGroup(groupDescription) -->
<xsd:element name="createGroup" type="tCreateGroup"/>
<xsd:complexType name="tCreateGroup">
    <xsd:complexContent>
        <xsd:extension base="tActivity">
            <xsd:sequence>
                <xsd:element name="input" type="tInputCreateGroup"/>
                <xsd:element name="output" type="tOutputCreateGroup"
                    minOccurs="0"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputCreateGroup">
    <xsd:sequence>
        <xsd:element name="groupDescription" type="tFrom"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputCreateGroup">
    <xsd:sequence>
        <xsd:element name="groupId" type="tTo"/>
    </xsd:sequence>
</xsd:complexType>
<!-- groupId[] getGroups(groupDescription?) -->
<xsd:element name="getGroups" type="tGetGroups"/>
<xsd:complexType name="tGetGroups">
    <xsd:complexContent>
        <xsd:extension base="tActivity">
            <xsd:sequence>
                <xsd:element name="input" type="tInputGetGroups"
                    minOccurs="0"/>
                <xsd:element name="output" type="tOutputGetGroups"
                    minOccurs="0"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputGetGroups">
    <xsd:sequence>
```

```
                <xsd:element name="groupDescription" type="tFrom" minOccurs="0"/>
        </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputGetGroups">
        <xsd:sequence>
                <xsd:element name="groupIdArray" type="tTo"/>
        </xsd:sequence>
</xsd:complexType>
<!-- groupGescription getGroupDescription(groupId) -->
<xsd:element name="getGroupDescription" type="tGetGroupDescription"/>
<xsd:complexType name="tGetGroupDescription">
        <xsd:complexContent>
                <xsd:extension base="tActivity">
                        <xsd:sequence>
                                <xsd:element name="input" type="tInputGetGroupDescription"/>
                                <xsd:element name="output"
                                    type="tOutputGetGroupDescription" minOccurs="0"/>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputGetGroupDescription">
        <xsd:sequence>
                <xsd:element name="groupId" type="tFrom"/>
        </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="tOutputGetGroupDescription">
        <xsd:sequence>
                <xsd:element name="groupDescription" type="tTo"/>
        </xsd:sequence>
</xsd:complexType>
<!-- void joinGroup(groupId, credentials) -->
<xsd:element name="joinGroup" type="tJoinGroup"/>
<xsd:complexType name="tJoinGroup">
        <xsd:complexContent>
                <xsd:extension base="tActivity">
                        <xsd:sequence>
                                <xsd:element name="input" type="tInputJoinGroup"/>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputJoinGroup">
        <xsd:sequence>
                <xsd:element name="groupId" type="tFrom"/>
                <xsd:element name="credentials" type="tFrom"/>
        </xsd:sequence>
</xsd:complexType>
<!-- void leaveGroup(groupId) -->
<xsd:element name="leaveGroup" type="tLeaveGroup"/>
<xsd:complexType name="tLeaveGroup">
        <xsd:complexContent>
                <xsd:extension base="tActivity">
                        <xsd:sequence>
                                <xsd:element name="input" type="tInputLeaveGroup"/>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputLeaveGroup">
        <xsd:sequence>
                <xsd:element name="groupId" type="tFrom"/>
        </xsd:sequence>
</xsd:complexType>
<!-- groupId[] getIncludingGroups() -->
<xsd:element name="getIncludingGroups" type="tGetIncludingGroups"/>
<xsd:complexType name="tGetIncludingGroups">
        <xsd:complexContent>
                <xsd:extension base="tActivity">
                        <xsd:sequence>
```

```
                                        <xsd:element name="output" type="tOutputGetIncludingGroups"
                                            minOccurs="0"/>
                                </xsd:sequence>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="tOutputGetIncludingGroups">
                <xsd:sequence>
                        <xsd:element name="groupIdArray" type="tTo"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- groupId getPublishingGroup(id?) -->
        <xsd:element name="getPublishingGroup" type="tGetPublishingGroup"/>
        <xsd:complexType name="tGetPublishingGroup">
                <xsd:complexContent>
                        <xsd:extension base="tActivity">
                                <xsd:sequence>
                                        <xsd:element name="input" type="tInputGetPublishingGroup"
                                            minOccurs="0"/>
                                        <xsd:element name="output" type="tOutputGetPublishingGroup"
                                            minOccurs="0"/>
                                </xsd:sequence>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="tInputGetPublishingGroup">
                <xsd:sequence>
                        <xsd:element name="id" type="tFrom"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="tOutputGetPublishingGroup">
                <xsd:sequence>
                        <xsd:element name="groupId" type="tTo"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- Service Management Primitives -->
        <!-- <groupServiceId, peerServiceId> publish(groupId, serviceContract) -->
        <xsd:element name="publish" type="tPublish"/>
        <xsd:complexType name="tPublish">
                <xsd:complexContent>
                        <xsd:extension base="tActivity">
                                <xsd:sequence>
                                        <xsd:element name="input" type="tInputPublish"/>
                                        <xsd:element name="output" type="tOutputPublish"
                                            minOccurs="0"/>
                                </xsd:sequence>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="tInputPublish">
                <xsd:sequence>
                        <xsd:element name="groupId" type="tFrom"/>
                        <xsd:element name="serviceContract" type="tFrom"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="tOutputPublish">
                <xsd:sequence>
                        <xsd:element name="groupServiceId" type="tTo"/>
                        <xsd:element name="peerServiceId" type="tTo"/>
                </xsd:sequence>
        </xsd:complexType>
        <!-- void unpublish(peerServiceId) -->
        <xsd:element name="unpublish" type="tUnpublish"/>
        <xsd:complexType name="tUnpublish">
                <xsd:complexContent>
                        <xsd:extension base="tActivity">
                                <xsd:sequence>
                                        <xsd:element name="input" type="tInputUnpublish"/>
                                </xsd:sequence>
                        </xsd:extension>
```

```
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="tInputUnpublish">
        <xsd:sequence>
            <xsd:element name="peerServiceId" type="tFrom"/>
        </xsd:sequence>
    </xsd:complexType>
    <!-- <groupId, groupServiceId, peerServiceId>[] getServices(groupId?, peerId?, serviceContract?,
        maxResults?, credentials) -->
    <xsd:element name="getServices" type="tGetServices"/>
    <xsd:complexType name="tGetServices">
        <xsd:complexContent>
            <xsd:extension base="tActivity">
                <xsd:sequence>
                    <xsd:element name="input" type="tInputGetServices"/>
                    <xsd:element name="output" type="tOutputGetServices"
                        minOccurs="0"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="tInputGetServices">
        <xsd:sequence>
            <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
            <xsd:element name="peerId" type="tFrom" minOccurs="0"/>
            <xsd:element name="serviceContract" type="tFrom" minOccurs="0"/>
            <xsd:element name="maxResults" type="tFrom" minOccurs="0"/>
            <xsd:element name="credentials" type="tFrom"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="tOutputGetServices">
        <xsd:sequence>
            <!-- EDIT jbuchhol ; getServices must return an array of
                <groupId,groupServiceId,peerServiceId>
            <xsd:element name="groupId" type="tTo" minOccurs="0"/>
            <xsd:element name="groupServiceId" type="tTo" minOccurs="0"/>
            <xsd:element name="peerServiceId" type="tTo" minOccurs="0"/>-->
            <xsd:element name="tripleArray" type="tTo"/>
        </xsd:sequence>
    </xsd:complexType>
    <!-- serviceContract getServiceContract(id) -->
    <xsd:element name="getServiceContract" type="tGetServiceContract"/>
    <xsd:complexType name="tGetServiceContract">
        <xsd:complexContent>
            <xsd:extension base="tActivity">
                <xsd:sequence>
                    <xsd:element name="input" type="tInputGetServiceContract"/>
                    <xsd:element name="output" type="tOutputGetServiceContract"
                        minOccurs="0"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="tInputGetServiceContract">
        <xsd:sequence>
            <xsd:element name="id" type="tFrom"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="tOutputGetServiceContract">
        <xsd:sequence>
            <xsd:element name="serviceContract" type="tTo"/>
        </xsd:sequence>
    </xsd:complexType>
    <!-- sessionId startSession(serviceId) -->
    <xsd:element name="startSession" type="tStartSession"/>
    <xsd:complexType name="tStartSession">
        <xsd:complexContent>
            <xsd:extension base="tActivity">
                <xsd:sequence>
                    <xsd:element name="input" type="tInputStartSession"/>
```

```
                                        <xsd:element name="output" type="tOutputStartSession"
                                            minOccurs="0"/>
                                    </xsd:sequence>
                                </xsd:extension>
                        </xsd:complexContent>
                </xsd:complexType>
                <xsd:complexType name="tInputStartSession">
                        <xsd:sequence>
                                <xsd:element name="serviceId" type="tFrom"/>
                        </xsd:sequence>
475             </xsd:complexType>
                <xsd:complexType name="tOutputStartSession">
                        <xsd:sequence>
                                <xsd:element name="sessionId" type="tTo"/>
                        </xsd:sequence>
                </xsd:complexType>
                <!-- Message Management Primitives -->
                <!-- output? invoke(entityId, operationName, input?) -->
                <xsd:element name="invoke" type="tInvoke"/>
                <xsd:complexType name="tInvoke">
485                     <xsd:complexContent>
                                <xsd:extension base="tActivity">
                                        <xsd:sequence>
                                                <xsd:element name="input" type="tInputInvoke"/>
                                                <xsd:element name="output" type="tOutputInvoke"
                                                    minOccurs="0"/>
                                        </xsd:sequence>
                                </xsd:extension>
                        </xsd:complexContent>
                </xsd:complexType>
                <xsd:complexType name="tInputInvoke">
495                     <xsd:sequence>
                                <xsd:element name="entityId" type="tFrom"/>
                                <xsd:element name="operationName" type="tFrom"/>
                                <xsd:element name="input" type="tFrom" minOccurs="0"/>
                        </xsd:sequence>
                </xsd:complexType>
                <xsd:complexType name="tOutputInvoke">
                        <xsd:sequence>
                                <xsd:element name="output" type="tTo"/>
                        </xsd:sequence>
505             </xsd:complexType>
                <!-- <callerId, input?> receiveMessage(groupId?, operationName) -->
                <xsd:element name="receiveMessage" type="tReceiveMessage"/>
                <xsd:complexType name="tReceiveMessage">
                        <xsd:complexContent>
                                <xsd:extension base="tActivity">
                                        <xsd:sequence>
                                                <xsd:element name="input" type="tInputReceiveMessage"/>
                                                <xsd:element name="output" type="tOutputReceiveMessage"/>
                                        </xsd:sequence>
515                             </xsd:extension>
                        </xsd:complexContent>
                </xsd:complexType>
                <xsd:complexType name="tInputReceiveMessage">
                        <xsd:sequence>
                                <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
                                <xsd:element name="operationName" type="tFrom"/>
                        </xsd:sequence>
                </xsd:complexType>
                <xsd:complexType name="tOutputReceiveMessage">
525                     <xsd:sequence>
                                <xsd:element name="callerId" type="tFrom"/>
                                <xsd:element name="input" type="tFrom" minOccurs="0"/>
                        </xsd:sequence>
                </xsd:complexType>
                <!-- void reply(callerId, operationName, output?, faultName?) -->
                <xsd:element name="reply" type="tReply"/>
                <xsd:complexType name="tReply">
                        <xsd:complexContent>
```

```
                <xsd:extension base="tActivity">
                    <xsd:sequence>
                        <xsd:element name="input" type="tInputReply"/>
                    </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="tInputReply">
            <xsd:sequence>
                <xsd:element name="callerId" type="tFrom"/>
                <xsd:element name="operationName" type="tFrom"/>
                <xsd:element name="output" type="tFrom" minOccurs="0"/>
                <xsd:element name="faultName" type="tFrom" minOccurs="0"/>
            </xsd:sequence>
        </xsd:complexType>
        <!-- Event Management Primitives -->
        <!-- void event(groupId?, eventName, input?) -->
        <xsd:element name="event" type="tEvent"/>
        <xsd:complexType name="tEvent">
            <xsd:complexContent>
                <xsd:extension base="tActivity">
                    <xsd:sequence>
                        <xsd:element name="input" type="tInputEvent"/>
                    </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="tInputEvent">
            <xsd:sequence>
                <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
                <xsd:element name="eventName" type="tFrom"/>
                <xsd:element name="input" type="tFrom" minOccurs="0"/>
            </xsd:sequence>
        </xsd:complexType>
        <!-- <callerId, input?> receiveEvent(groupId?, eventName) -->
        <xsd:element name="receiveEvent" type="tReceiveEvent"/>
        <xsd:complexType name="tReceiveEvent">
            <xsd:complexContent>
                <xsd:extension base="tActivity">
                    <xsd:sequence>
                        <xsd:element name="input" type="tInputReceiveEvent"/>
                        <xsd:element name="output" type="tOutputReceiveEvent"/>
                    </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="tInputReceiveEvent">
            <xsd:sequence>
                <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
                <xsd:element name="eventName" type="tFrom"/>
            </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="tOutputReceiveEvent">
            <xsd:sequence>
                <xsd:element name="callerId" type="tFrom"/>
                <xsd:element name="input" type="tFrom" minOccurs="0"/>
            </xsd:sequence>
        </xsd:complexType>
        <!-- void subscribe(eventName?, groupId?) -->
        <xsd:element name="subscribe" type="tSubscribe"/>
        <xsd:complexType name="tSubscribe">
            <xsd:complexContent>
                <xsd:extension base="tActivity">
                    <xsd:sequence>
                        <xsd:element name="input" type="tInputSubscribe"
                            minOccurs="0"/>
                    </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
```

```
<xsd:complexType name="tInputSubscribe">
    <xsd:sequence>
        <xsd:element name="eventName" type="tFrom" minOccurs="0"/>
        <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
<!-- void unsubscribe(eventName?, groupId?) -->
<xsd:element name="unsubscribe" type="tUnsubscribe"/>
<xsd:complexType name="tUnsubscribe">
    <xsd:complexContent>
        <xsd:extension base="tActivity">
            <xsd:sequence>
                <xsd:element name="input" type="tInputUnsubscribe"
                    minOccurs="0"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="tInputUnsubscribe">
    <xsd:sequence>
        <xsd:element name="eventName" type="tFrom" minOccurs="0"/>
        <xsd:element name="groupId" type="tFrom" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
<!-- End of primitives definition. -->
<xsd:element name="empty" type="tEmpty"/>
<xsd:complexType name="tEmpty">
    <xsd:complexContent>
        <xsd:extension base="tActivity"/>
    </xsd:complexContent>
</xsd:complexType>
<xsd:element name="assign" type="tAssign"/>
<xsd:complexType name="tAssign">
    <xsd:complexContent>
        <xsd:extension base="tActivity">
            <xsd:sequence>
                <xsd:element ref="copy" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:element name="copy" type="tCopy"/>
<xsd:complexType name="tCopy">
    <xsd:complexContent>
        <xsd:extension base="tExtensibleElements">
            <xsd:sequence>
                <xsd:element ref="from"/>
                <xsd:element ref="to"/>
            </xsd:sequence>
            <xsd:attribute name="keepSrcElementName" type="tBoolean"
                use="optional" default="no"/>
            <xsd:attribute name="ignoreMissingFromData" type="tBoolean"
                use="optional" default="no"/>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="tBoolean">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="yes"/>
        <xsd:enumeration value="no"/>
    </xsd:restriction>
</xsd:simpleType>
<xsd:element name="from" type="tFrom"/>
<xsd:complexType name="tFrom" mixed="true">
    <xsd:sequence>
        <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:choice minOccurs="0">
            <xsd:element ref="literal"/>
```

```
                        <xsd:element ref="query"/>
                    </xsd:choice>
                </xsd:sequence>
                <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
                <xsd:attribute name="variable" type="VariableName"/>
                <xsd:attribute name="part" type="xsd:NCName"/>
                <xsd:anyAttribute namespace="##other" processContents="lax"/>
            </xsd:complexType>
            <xsd:element name="literal" type="tLiteral"/>
            <xsd:complexType name="tLiteral" mixed="true">
                <xsd:sequence>
                    <xsd:any namespace="##any" processContents="lax" minOccurs="0"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:element name="query" type="tQuery"/>
            <xsd:complexType name="tQuery" mixed="true">
                <xsd:sequence>
                    <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="queryLanguage" type="xsd:anyURI"/>
                <xsd:anyAttribute namespace="##other" processContents="lax"/>
            </xsd:complexType>
            <xsd:element name="to" type="tTo"/>
            <xsd:complexType name="tTo" mixed="true">
                <xsd:sequence>
                    <xsd:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
                    <xsd:any namespace="##other" processContents="lax" minOccurs="0"
                        maxOccurs="unbounded"/>
                    <xsd:element ref="query" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
                <xsd:attribute name="variable" type="VariableName"/>
                <xsd:attribute name="part" type="xsd:NCName"/>
                <xsd:anyAttribute namespace="##other" processContents="lax"/>
            </xsd:complexType>
            <xsd:element name="wait" type="tWait"/>
            <xsd:complexType name="tWait">
                <xsd:complexContent>
                    <xsd:extension base="tActivity">
                        <xsd:sequence>
                            <xsd:choice>
                                <xsd:element ref="for" minOccurs="0"/>
                                <xsd:element ref="until" minOccurs="0"/>
                                <xsd:element ref="repeatEvery" minOccurs="0"/>
                            </xsd:choice>
                            <!--<xsd:element ref="repeatEvery" minOccurs="0"/>-->
                            <!-- The repeatEvery element can only be defined for
                                informationHandler branches -->
                        </xsd:sequence>
                    </xsd:extension>
                </xsd:complexContent>
            </xsd:complexType>
            <xsd:element name="for" type="tDuration-expr"/>
            <xsd:element name="until" type="tDeadline-expr"/>
            <xsd:element name="repeatEvery" type="tDuration-expr"/>
            <xsd:complexType name="tDuration-expr" mixed="true">
                <xsd:complexContent mixed="true">
                    <xsd:extension base="tExpression"/>
                </xsd:complexContent>
            </xsd:complexType>
            <xsd:complexType name="tDeadline-expr" mixed="true">
                <xsd:complexContent mixed="true">
                    <xsd:extension base="tExpression"/>
                </xsd:complexContent>
            </xsd:complexType>
            <xsd:complexType name="tExpression" mixed="true">
                <xsd:sequence>
                    <xsd:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
```

```
        <xsd:anyAttribute namespace="##other" processContents="lax"/>
    </xsd:complexType>
    <xsd:element name="throw" type="tThrow"/>
    <xsd:complexType name="tThrow">
        <xsd:complexContent>
            <xsd:extension base="tActivity">
                <xsd:sequence>
                    <xsd:element name="input" type="tInputThrow"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="tInputThrow">
        <xsd:sequence>
            <xsd:element name="faultName" type="tFrom"/>
            <xsd:element name="faultVariable" type="tFrom" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="catch" type="tCatch"/>
    <xsd:complexType name="tCatch">
        <xsd:complexContent>
            <xsd:extension base="tActivityContainer">
                <xsd:sequence>
                    <xsd:element name="input" type="tInputCatch"/>
                    <xsd:element name="output" type="tOutputCatch"
                        minOccurs="0"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="tInputCatch">
        <xsd:sequence>
            <xsd:element name="faultName" type="tFrom"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="tOutputCatch">
        <xsd:sequence>
            <xsd:element name="faultVariable" type="tTo"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="tActivityContainer">
        <xsd:complexContent>
            <xsd:extension base="tExtensibleElements">
                <xsd:sequence>
                    <xsd:group ref="activity"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:element name="catchAll" type="tCatchAll"/>
    <xsd:complexType name="tCatchAll">
        <xsd:complexContent>
            <xsd:extension base="tActivityContainer">
                <xsd:sequence>
                    <xsd:element name="output" type="tOutputCatchAll"
                        minOccurs="0"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="tOutputCatchAll">
        <xsd:sequence>
            <xsd:element name="faultName" type="tTo"/>
            <xsd:element name="faultVariable" type="tTo" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="exit" type="tExit"/>
    <xsd:complexType name="tExit">
        <xsd:complexContent>
            <xsd:extension base="tActivity"/>
```

```
805             </xsd:complexContent>
        </xsd:complexType>
        <!-- End of basic_commands definition. -->
        <!-- Start of structured_commands definition. -->
        <xsd:element name="sequence" type="tSequence"/>
        <xsd:complexType name="tSequence">
            <xsd:complexContent>
                <xsd:extension base="tActivity">
                    <xsd:sequence>
                        <xsd:group ref="activity" maxOccurs="unbounded"/>
815                 </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="flow" type="tFlow"/>
        <xsd:complexType name="tFlow">
            <xsd:complexContent>
                <xsd:extension base="tActivity">
                    <xsd:sequence>
                        <xsd:group ref="activity" maxOccurs="unbounded"/>
825                 </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="while" type="tWhile"/>
        <xsd:complexType name="tWhile">
            <xsd:complexContent>
                <xsd:extension base="tActivity">
                    <xsd:sequence>
                        <xsd:element ref="condition"/>
835                     <xsd:group ref="activity"/>
                    </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="condition" type="tBoolean-expr"/>
        <xsd:complexType name="tBoolean-expr" mixed="true">
            <xsd:complexContent mixed="true">
                <xsd:extension base="tExpression"/>
            </xsd:complexContent>
845     </xsd:complexType>
        <xsd:element name="repeatUntil" type="tRepeatUntil"/>
        <xsd:complexType name="tRepeatUntil">
            <xsd:complexContent>
                <xsd:extension base="tActivity">
                    <xsd:sequence>
                        <xsd:group ref="activity"/>
                        <xsd:element ref="condition"/>
                    </xsd:sequence>
                </xsd:extension>
855         </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="if" type="tIf"/>
        <xsd:complexType name="tIf">
            <xsd:complexContent>
                <xsd:extension base="tActivity">
                    <xsd:sequence>
                        <xsd:element ref="condition"/>
                        <xsd:group ref="activity"/>
                        <xsd:element ref="else" minOccurs="0"/>
865                 </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="else" type="tActivityContainer"/>
        <xsd:element name="pick" type="tPick"/>
        <xsd:complexType name="tPick">
            <xsd:complexContent>
                <xsd:extension base="tActivity">
                    <xsd:sequence>
```

```
                                        <xsd:element ref="onMessage" minOccurs="0"
                                            maxOccurs="unbounded"/>
                                        <xsd:element ref="onEvent" minOccurs="0"
                                            maxOccurs="unbounded"/>
                                        <xsd:element ref="onAlarm" minOccurs="0"
                                            maxOccurs="unbounded"/>
                                </xsd:sequence>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="onMessage" type="tOnMessage"/>
        <xsd:complexType name="tOnMessage">
                <xsd:complexContent>
                        <xsd:extension base="tActivityContainer">
                                <xsd:sequence>
                                        <xsd:element ref="receiveMessage"/>
                                </xsd:sequence>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="onEvent" type="tOnEvent"/>
        <xsd:complexType name="tOnEvent">
                <xsd:complexContent>
                        <xsd:extension base="tActivityContainer">
                                <xsd:sequence>
                                        <xsd:element ref="receiveEvent"/>
                                </xsd:sequence>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="onAlarm" type="tOnAlarm"/>
        <xsd:complexType name="tOnAlarm">
                <xsd:complexContent>
                        <xsd:extension base="tActivityContainer">
                                <xsd:sequence>
                                        <xsd:element ref="wait"/>
                                </xsd:sequence>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="informationHandler" type="tInformationHandler"/>
        <xsd:complexType name="tInformationHandler">
                <xsd:complexContent>
                        <xsd:extension base="tActivity">
                                <xsd:sequence>
                                        <xsd:group ref="activity"/>
                                        <xsd:element ref="onMessage" minOccurs="0"
                                            maxOccurs="unbounded"/>
                                        <xsd:element ref="onEvent" minOccurs="0"
                                            maxOccurs="unbounded"/>
                                        <xsd:element ref="onAlarm" minOccurs="0"
                                            maxOccurs="unbounded"/>
                                </xsd:sequence>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <xsd:element name="faultHandler" type="tFaultHandler"/>
        <xsd:complexType name="tFaultHandler">
                <xsd:complexContent>
                        <xsd:extension base="tActivity">
                                <xsd:sequence>
                                        <xsd:group ref="activity"/>
                                        <xsd:element ref="catch" minOccurs="0"
                                            maxOccurs="unbounded"/>
                                        <xsd:element ref="catchAll" minOccurs="0"/>
                                </xsd:sequence>
                        </xsd:extension>
                </xsd:complexContent>
        </xsd:complexType>
        <!-- End of structured_commands definition. -->
```

```
<!-- Begin of data types definition -->
<xsd:element name="Credential" type="Credential"/>
<xsd:complexType name="Credential">
    <xsd:sequence>
        <xsd:element name="smeppkey" type="tSmeppKey"/>
        <xsd:element name="groupkeys" type="tSmeppKey" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
    <xsd:attribute name="variable" type="VariableName"/>
    <xsd:attribute name="part" type="xsd:NCName"/>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>
<xsd:complexType name="tSmeppKey">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="pwd" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:element name="GroupDescription" type="GroupDescription"/>
<xsd:complexType name="GroupDescription">
    <xsd:sequence>
        <xsd:element name="groupName" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
    <xsd:attribute name="variable" type="VariableName"/>
    <xsd:attribute name="part" type="xsd:NCName"/>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>
<xsd:element name="ID" type="ID"/>
<xsd:complexType name="ID">
    <xsd:sequence>
        <xsd:element name="peerId" type="PeerID" minOccurs="0"/>
        <xsd:element name="groupId" type="GroupID" minOccurs="0"/>
        <xsd:element name="sessionId" type="SessionID" minOccurs="0"/>
        <xsd:element name="groupServiceId" type="GroupServiceID" minOccurs="0"/>
        <xsd:element name="peerServiceId" type="PeerServiceID" minOccurs="0"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="PeerID">
    <xsd:sequence>
        <xsd:element name="opaquePID" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
    <xsd:attribute name="variable" type="VariableName"/>
    <xsd:attribute name="part" type="xsd:NCName"/>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>
<xsd:complexType name="GroupID">
    <xsd:sequence>
        <xsd:element name="opaqueGID" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
    <xsd:attribute name="variable" type="VariableName"/>
    <xsd:attribute name="part" type="xsd:NCName"/>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>
<xsd:complexType name="GroupServiceID">
    <xsd:sequence>
        <xsd:element name="opaqueGSID" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
    <xsd:attribute name="variable" type="VariableName"/>
    <xsd:attribute name="part" type="xsd:NCName"/>
    <xsd:anyAttribute namespace="##other" processContents="lax"/>
</xsd:complexType>
<xsd:complexType name="PeerServiceID">
    <xsd:sequence>
        <xsd:element name="opaquePSID" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
```

```
            <xsd:attribute name="variable" type="VariableName"/>
            <xsd:attribute name="part" type="xsd:NCName"/>
            <xsd:anyAttribute namespace="##other" processContents="lax"/>
        </xsd:complexType>
        <xsd:complexType name="SessionID">
            <xsd:sequence>
                <xsd:element name="opaqueSSID" type="xsd:string"/>
            </xsd:sequence>
            <xsd:attribute name="expressionLanguage" type="xsd:anyURI"/>
            <xsd:attribute name="variable" type="VariableName"/>
            <xsd:attribute name="part" type="xsd:NCName"/>
            <xsd:anyAttribute namespace="##other" processContents="lax"/>
        </xsd:complexType>
        <!-- End of data types definition -->
</xsd:schema>
```

Appendix B

Application example

B.1 SMoL code

B.1.1 TempReaderPeer's code

```
<?xml version="1.0" encoding="UTF-8"?>
<process targetNamespace="" name="NCName" xsi:schemaLocation="http://SMoL.SMEPP.org
    SMoLv21.xsd" xmlns="http://SMoL.SMEPP.org"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<variables>
        <variable name="myCredentials" type="Credential"/>
        <variable name="tempGroupId" type="GroupID"/>
        <variable name="temp5s" type="Input"/>
        <variable name="temp10s" type="Input"/>
        <variable name="temp" type="Input"/>
</variables>

<sequence>
        <assign>
                <copy>
                        <from>
                                <literal>
                                        <Credential>
                                                <smeppkey>
                                                        <name>SMEPPKEY</name>
                                                        <pwd>SMEPPPWD</pwd>
                                                </smeppkey>
                                                <groupkeys>
                                                        <name>TempReaderGroup</name>
                                                        <pwd>MyGroupNamePassword</pwd>
                                                </groupkeys>
                                        </Credential>
                                </literal>
                        </from>
                        <to variable="myCredentials"/>
                </copy>
        </assign>

        <newPeer>
                <input>
                        <credentials variable="myCredentials"/>
                </input>
        </newPeer>

        <createGroup>
                <input>
                        <groupDescription>
                                <literal>
                                        <GroupDescription>
                                                <groupName>TempReaderGroup</groupName>
```

```
                                        </GroupDescription>
                                </literal>
                        </groupDescription>
48              </input>
                <output>
                        <groupId variable="tempGroupId"/>
                </output>
        </createGroup>

        <flow>
                <while>
                        <condition>true</condition>
                        <sequence>
58                              <assign>
                                <copy>
                                        <from>
                                                <literal>
                                                        <Input>new
                                                                Long(System.currentTimeMillis())</Input>
                                                </literal>
                                        </from>
                                        <to variable="temp5s"></to>
                                </copy>
                                </assign>
68
                                <event>
                                  <input>
                                   <groupId variable="tempGroupId"/>
                                   <eventName><literal><String>temp5s</String></literal></eventName>
                                   <input><literal><Input>TempReaderPeerBDataTable.temp</Input></literal></input>
                                  </input>
                                </event>
                                <wait>
                                        <for>PT5S</for>
78                              </wait>
                        </sequence>
                </while>
                <while>
                        <condition>true</condition>
                        <sequence>
                                <assign>
                                <copy>
                                        <from>
                                                <literal>
88                                                      <Input>new
                                                                Long(System.currentTimeMillis())</Input>
                                                </literal>
                                        </from>
                                        <to variable="temp10s"></to>
                                </copy>
                                </assign>

                                <event>
                                 <input>
                                  <groupId variable="tempGroupId"/>
98                                <eventName><literal><String>temp10s</String></literal></eventName>
                                  <input><literal><Input>TempReaderPeerBDataTable.temp</Input></literal></input>
                                 </input>
                                </event>
                                <wait>
                                        <for>PT10S</for>
                                </wait>
                        </sequence>
                </while>
        </flow>
108
</sequence>

</process>
```

B.1.2 ClientPeer1's code

```
<?xml version="1.0" encoding="UTF-8"?>
<process targetNamespace="" name="NCName" xsi:schemaLocation="http://SMoL.SMEPP.org
    SMoLv21.xsd" xmlns="http://SMoL.SMEPP.org"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<variables>
    <variable name="myCredentials" type="Credential"/>
    <variable name="tempGroupId" type="GroupID"/>
    <variable name="temp5s" type="Input"/>
    <variable name="temp10s" type="Input"/>
    <variable name="temp" type="Input"/>
    <variable name="gidArray" type="GroupIDArray"/>
    <variable name="caller_event" type="CallerID"/>
</variables>

<sequence>
    <assign>
        <copy>
            <from>
                <literal>
                    <Credential>
                        <smeppkey>
                            <name>SMEPPKEY</name>
                            <pwd>SMEPPPWD</pwd>
                        </smeppkey>
                        <groupkeys>
                            <name>TempReaderGroup</name>
                            <pwd>MyGroupNamePassword</pwd>
                        </groupkeys>
                    </Credential>
                </literal>
            </from>
            <to variable="myCredentials"/>
        </copy>
    </assign>

    <newPeer>
        <input>
            <credentials variable="myCredentials"/>
        </input>
    </newPeer>

    <getGroups>
        <input>
            <groupDescription>
                <literal>
                    <GroupDescription>
                        <groupName>TempReaderGroup</groupName>
                    </GroupDescription>
                </literal>
            </groupDescription>
        </input>
        <output>
            <groupIdArray variable="gidArray"/>
        </output>
    </getGroups>

        <joinGroup>
        <input>
            <groupId variable="gidArray"><query>/gidArray[0]</query></groupId>
            <credentials variable="myCredentials"/>
        </input>
    </joinGroup>

    <subscribe>
        <input>
            <eventName><literal><String>temp5s</String></literal></eventName>
            <groupId variable="gidArray"><query>/gidArray[0]</query></groupId>
```

```
                </input>
        </subscribe>
69
        <informationHandler>
                <sequence>
                        <wait>
                                <for>PT1H</for>
                        </wait>
                        <unsubscribe></unsubscribe>
                </sequence>
                <onEvent>
                        <!-- use the event load for smtgh -->
79                      <sequence>
                                <empty></empty>
                        </sequence>
                        <receiveEvent>
                                <input>
                                        <groupId
                                             variable="gidArray"><query>/gidArray[0]</query></groupId>
                                        <eventName><literal><String>temp5s</String></literal></eventName>
                                </input>
                                <output>
                                        <callerId variable="caller_event"/>
89                                      <input variable="temp10s"/>
                                </output>
                        </receiveEvent>
                </onEvent>
        </informationHandler>


</sequence>

</process>
```

B.1.3 ClientPeer2's code

```
<?xml version="1.0" encoding="UTF-8"?>
 2  <process targetNamespace="" name="NCName" xsi:schemaLocation="http://SMoL.SMEPP.org␣
        SMoLv21.xsd" xmlns="http://SMoL.SMEPP.org"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <variables>
            <variable name="myCredentials" type="Credential"/>
            <variable name="tempGroupId" type="GroupID"/>
            <variable name="temp" type="Input"/>
            <variable name="gidArray" type="GroupIDArray"/>
            <variable name="psid" type="PeerServiceID"/>
            <variable name="gsid" type="GroupServiceID"/>
    </variables>
12
    <sequence>
            <assign>
                    <copy>
                            <from>
                                    <literal>
                                            <Credential>
                                                    <smeppkey>
                                                            <name>SMEPPKEY</name>
                                                            <pwd>SMEPPPWD</pwd>
22                                                  </smeppkey>
                                                    <groupkeys>
                                                            <name>TempReaderGroup</name>
                                                            <pwd>MyGroupNamePassword</pwd>
                                                    </groupkeys>
                                            </Credential>
                                    </literal>
                            </from>
                            <to variable="myCredentials"/>
```

```
                </copy>
        </assign>

        <newPeer>
                <input>
                        <credentials variable="myCredentials"/>
                </input>
        </newPeer>

        <getGroups>
                <input>
                        <groupDescription>
                                <literal>
                                        <GroupDescription>
                                                <groupName>TempReaderGroup</groupName>
                                        </GroupDescription>
                                </literal>
                        </groupDescription>
                </input>
                <output>
                        <groupIdArray variable="gidArray"/>
                </output>
        </getGroups>

                <joinGroup>
                <input>
                        <groupId variable="gidArray"><query>/gidArray[0]</query></groupId>
                        <credentials variable="myCredentials"/>
                </input>
        </joinGroup>

        <publish>
                <input>
                        <groupId variable="gidArray"><query>/gidArray[0]</query></groupId>
                        <serviceContract><literal>./ClientService2.xml</literal></serviceContract>
                </input>
                <output>
                        <groupServiceId variable="gsid"></groupServiceId>
                        <peerServiceId variable="psid"></peerServiceId>
                </output>
        </publish>
        <invoke>
                <input>
                        <entityId variable="psid"/>
                        <operationName><literal><String>monitor</String></literal></operationName>
                        <input><literal><Input>ClientPeer2BDataTable.gidArray[0]</Input></literal></input>
                </input>
                <output>
                        <output variable ="temp"/>
                </output>
        </invoke>

</sequence>

</process>
```

B.1.4 ClientService's code

```
<?xml version="1.0" encoding="UTF-8"?>
<Contract xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3.org/2001/XMLSchema_Contract.xsd"
xmlns:sign="http://www.w3.org/ns/wsdl" xmlns:smol="http://SMoL.SMEPP.org"
xmlns="http://www.w3.org/2001/XMLSchema">
        <Signature>
                <service name="ClientService2" type="state-less" category="unknown"/>
                <sign:description>
                        <sign:types/>
                        <sign:interface name="ClientService2">
```

```
                        <sign:operation name="monitor">
                            <sign:input messageLabel="gr" element="groupId"/>
                            <sign:output messageLabel="temp" element="Input"/>
                        </sign:operation>
                    </sign:interface>
                </sign:description>
            </Signature>

            <Grounding>
                <!--<Parameters>
                    <param name="protocol" type="xsi:string">RMI</param>
                    <param name="encryptation" type="xsi:string">MD5</param>
                </Parameters>-->
            </Grounding>

            <Behavior>
            <smol:process targetNamespace="" name="NCName"
                xsi:schemaLocation="http://SMoL.SMEPP.org SMoLv21.xsd"
                xmlns="http://SMoL.SMEPP.org"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                <smol:variables>
                    <smol:variable name="callerId" type="CallerID"/>
                    <smol:variable name="temp" type="Input"/>
                    <smol:variable name="caller_temp" type="CallerID"/>
                    <smol:variable name="temp10s" type="Input"/>
                    <smol:variable name="gid" type="GroupID"/>
                </smol:variables>

                <smol:sequence>

                <smol:receiveMessage>
                    <smol:input>
<smol:operationName><smol:literal><smol:String>monitor</smol:String></smol:literal></smol:operationName>
                    </smol:input>
                    <smol:output>
                        <smol:callerId variable="callerId"/>
                        <smol:input variable="temp"/>
                    </smol:output>
                </smol:receiveMessage>

                    <smol:getPublishingGroup>
                    <smol:output>
                        <smol:groupId variable="gid"/>
                    </smol:output>
                </smol:getPublishingGroup>

                <smol:subscribe>
                    <smol:input>
                        <smol:eventName><literal><String>temp10s</String></literal></smol:eventName>
                        <smol:groupId variable="gid"/>
                    </smol:input>
                </smol:subscribe>

                    <smol:informationHandler>
                <smol:sequence>
                    <smol:wait>
                        <smol:for>PT30S</smol:for>
                    </smol:wait>
                    <smol:unsubscribe>
                        <smol:input>
                        <smol:eventName><literal><String>temp10s</String></literal></smol:eventName>
                        <smol:groupId variable="gid"/>
                        </smol:input>
                    </smol:unsubscribe>
                </smol:sequence>
                <smol:onEvent>
                    <sequence><empty></empty></sequence>
                    <!-- use the event load for smtgh -->
                    <smol:receiveEvent>
                        <smol:input>
```

```
                                <smol:groupId variable="gid"/>
                                <smol:eventName><literal><String>temp10s</String></literal></smol:eventName>
                                </smol:input>
                                <smol:output>
                                        <smol:callerId variable="caller_temp"/>
                                        <smol:input variable="temp10s"/>
                                </smol:output>
                        </smol:receiveEvent>
                </smol:onEvent>
        </smol:informationHandler>

                        <smol:reply>
                        <smol:input>
                        <smol:callerId variable="callerId"/>
<smol:operationName><smol:literal><smol:String>monitor</smol:String></smol:literal></smol:operationName>
                        <smol:output><literal><Input>new
                                String("anyOutputPossible")</Input></literal></smol:output>
                        </smol:input>
                </smol:reply>

                </smol:sequence>
        </smol:process>
        </Behavior>

</Contract>
```

B.2 Java code

B.2.1 TempReaderPeer's generated code

Root FaultHandler

```
import java.util.Vector;
import java.util.Iterator;
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class TempReaderPeerBRootFH extends FaultHandler{

public TempReaderPeerBRootFH instance;

public TempReaderPeerBRootFH(SMEPPCaller cal){
instance = this;

super.setCatches(new Vector());
super.getCatches().add(new Catch(true,null));
this.execute(cal);
if(this.getContainer() instanceof Peer){
this.execute(this);
}
}

public void execute(SMEPPCaller cal){
super.setCaller(cal);
}

public void execute(ThreadedCommand parent) {

//a temp Pair variable is declared in order to use primitive's return composed type (having more than one
    value)
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in
    the xml
Pair tempPair = null;
```

```
TempReaderPeerBDataTable.myCredentials = new Credential(new SMEPPKey("SMEPPKEY","
    SMEPPPWD"),
                new SMEPPKey[]{new SMEPPKey("TempReaderGroup","MyGroupNamePassword")});
peer.Primitives.getPrimitives().newPeer(TempReaderPeerBDataTable.myCredentials,instance);
TempReaderPeerBDataTable.tempGroupId =
                peer.Primitives.getPrimitives().createGroup(new GroupDescription("TempReaderGroup"),
                    instance);
TempReaderPeerBFlow11 TempReaderPeerBfl11 = new TempReaderPeerBFlow11(instance);
TempReaderPeerBfl11.start();
TempReaderPeerBFlow12 TempReaderPeerBfl12 = new TempReaderPeerBFlow12(instance);
TempReaderPeerBfl12.start();
try{TempReaderPeerBfl11.join();}
catch(InterruptedException e){e.printStackTrace();}
try{TempReaderPeerBfl12.join();}
catch(InterruptedException e){e.printStackTrace();}
}

synchronized public void forwardFault(Fault f) {
super.setStopped(true);
boolean catched = false;
Iterator it = super.getCatches().iterator();
if(!catched && it.hasNext()){
catched=true;
Catch c = (Catch) it.next();
if(c.isAll() || c.getFault().getName().equals(f.getName())){
System.out.println(f.getName()+" caught");
//catch main command
System.exit(1);

}
}
}
}
```

DataTable

```
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class TempReaderPeerBDataTable {

private static TempReaderPeerBDataTable instance;
public static Credential myCredentials;
public static GroupID tempGroupId;
public static Input temp5s;
public static Input temp10s;
public static Input temp;

private TempReaderPeerBDataTable(){

}

public static TempReaderPeerBDataTable getDataTable(){
if(instance == null){
instance = new TempReaderPeerBDataTable();
}
return instance;
}

}
```

Branch 1 flow

```
import java.util.Vector;
import java.util.Iterator;
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class TempReaderPeerBFlow11 extends Flow{

public TempReaderPeerBFlow11 instance;

public TempReaderPeerBFlow11(ThreadedCommand parent){
super.setUpperCmd(parent);
instance = this;
}

public void execute(ThreadedCommand parent) {

//a temp Pair variable is declared in order to use primitive's return composed type (having more than one
    value)
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in
    the xml
Pair tempPair = null;

while(true){
TempReaderPeerBDataTable.temp5s = new Input(new Long(System.currentTimeMillis()));
peer.Primitives.getPrimitives().event(TempReaderPeerBDataTable.tempGroupId,"temp5s",new Input(
    TempReaderPeerBDataTable.temp),instance);
try{Thread.sleep(parser.SMoLParser.translateXPathDuration("PT5S"));}
catch(InterruptedException e){e.printStackTrace();}

}
}

public void run(){
execute(super.getUpperCmd());
}
}
```

Branch 2 flow

```
import java.util.Vector;
import java.util.Iterator;
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class TempReaderPeerBFlow12 extends Flow{

public TempReaderPeerBFlow12 instance;

public TempReaderPeerBFlow12(ThreadedCommand parent){
super.setUpperCmd(parent);
instance = this;
}

public void execute(ThreadedCommand parent) {

//a temp Pair variable is declared in order to use primitive's return composed type (having more than one
    value)
```

```
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in
    the xml
Pair tempPair = null;

while(true){
TempReaderPeerBDataTable.temp10s = new Input(new Long(System.currentTimeMillis()));
peer.Primitives.getPrimitives().event(TempReaderPeerBDataTable.tempGroupId,"temp10s",new Input(
    TempReaderPeerBDataTable.temp),instance);
try{Thread.sleep(parser.SMoLParser.translateXPathDuration("PT10S"));}
catch(InterruptedException e){e.printStackTrace();}

}
}

public void run(){
execute(super.getUpperCmd());
}
}
```

B.2.2 ClientPeer1's generated code

Root FaultHander

```
import java.util.Vector;
import java.util.Iterator;
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class ClientPeer1BRootFH extends FaultHandler{

public ClientPeer1BRootFH instance;

public ClientPeer1BRootFH(SMEPPCaller cal){
instance = this;

super.setCatches(new Vector());
super.getCatches().add(new Catch(true,null));
this.execute(cal);
if(this.getContainer() instanceof Peer){
this.execute(this);
}
}

public void execute(SMEPPCaller cal){
super.setCaller(cal);
}

public void execute(ThreadedCommand parent) {

//a temp Pair variable is declared in order to use primitive's return composed type (having more than one
    value)
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in
    the xml
Pair tempPair = null;

ClientPeer1BDataTable.myCredentials = new Credential(new SMEPPKey("SMEPPKEY","SMEPPPWD"),
    new SMEPPKey[]{new SMEPPKey("TempReaderGroup","MyGroupNamePassword")});
peer.Primitives.getPrimitives().newPeer(ClientPeer1BDataTable.myCredentials,instance);
int i = 0;
while(i==0){
ClientPeer1BDataTable.gidArray = peer.Primitives.getPrimitives().getGroups(new GroupDescription("
    TempReaderGroup"),instance);
i = ClientPeer1BDataTable.gidArray.length;
try {Thread.sleep(1000);} catch (InterruptedException e) {e.printStackTrace();}
```

```
}
peer.Primitives.getPrimitives().joinGroup(ClientPeer1BDataTable.gidArray[0],ClientPeer1BDataTable.
    myCredentials,instance);
peer.Primitives.getPrimitives().subscribe("temp5s",ClientPeer1BDataTable.gidArray[0],instance);
ClientPeer1BIH11 ClientPeer1Bih11 = new ClientPeer1BIH11(instance);
}

synchronized public void forwardFault(Fault f) {
super.setStopped(true);
boolean catched = false;
Iterator it = super.getCatches().iterator();
if(!catched && it.hasNext()){
catched=true;
Catch c = (Catch) it.next();
if(c.isAll() || c.getFault().getName().equals(f.getName())){
System.out.println(f.getName()+"_caught");
//catch main command
System.exit(1);

}
}
}
}
```

DataTable

```
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class ClientPeer1BDataTable {

private static ClientPeer1BDataTable instance;
public static Credential myCredentials;
public static GroupID tempGroupId;
public static Input temp5s;
public static Input temp10s;
public static Input temp;
public static GroupID[] gidArray;
public static CallerID caller_event;

private ClientPeer1BDataTable(){

}

public static ClientPeer1BDataTable getDataTable(){
if(instance == null){
instance = new ClientPeer1BDataTable();
}
return instance;
}

}
```

InformationHandler root

```
import java.util.Vector;
import java.util.Iterator;
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
```

```
import utils.ids.*;

public class ClientPeer1BIH11 extends InformationHandler{

public ClientPeer1BIH11 instance;

public ClientPeer1BIH11(ThreadedCommand parent){
super.setUpperCmd(parent);
instance = this;
this.execute(super.getUpperCmd());
}

public void execute(ThreadedCommand parent) {

//a temp Pair variable is declared in order to use primitive's return composed type (having more than one
    value)
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in
    the xml
Pair tempPair = null;

ClientPeer1BIH11Branch1 ClientPeer1Bih11b1 = new ClientPeer1BIH11Branch1(instance);
ClientPeer1Bih11b1.start();
try{Thread.sleep(parser.SMoLParser.translateXPathDuration("PT1H"));}
catch(InterruptedException e){e.printStackTrace();}
peer.Primitives.getPrimitives().unsubscribe(instance);

super.setFinished();
}
}
```

InformationHandler branch 1

```
import lime.ILimeAgent;
import lime.LimeAgentMgr;
import lime.LimeThread;
import java.util.Vector;
import java.util.Iterator;
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class ClientPeer1BIH11Branch1 extends LimeThread implements ILimeAgent {

public InformationHandler instance;

public ClientPeer1BIH11Branch1(InformationHandler p){
instance = p;
}

public void run(){
//a temp Pair variable is declared in order to use primitive's return composed type (having more than one
    value)
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in
    the xml
Pair tempPair = null;

while(!instance.isFinished() || !instance.isStopped()){
//translation of the receiveEvent
tempPair = peer.Primitives.getPrimitives().receiveEvent(ClientPeer1BDataTable.gidArray[0],"temp5s",
    instance);
//check if a fault has been raised during the waiting period of this receiveEvent and returns if one has been
    raised
if(instance.isStopped()) return;
ClientPeer1BDataTable.temp10s = (Input) tempPair.getSecond();
ClientPeer1BDataTable.caller_event = (CallerID) tempPair.getFirst();
```

```
//check if the informationHandler main command isn't terminated or if a fault was raised
if(instance.isFinished() || instance.isStopped()) return;

//starts the thread which executes the receiveEvent's command
ClientPeer1BIH11Branch1Cmd ClientPeer1Bih11b1cmd = new ClientPeer1BIH11Branch1Cmd(instance);
ClientPeer1Bih11b1cmd.start();
}
}

public LimeAgentMgr getMgr() {
return getParent().getMgr();
}
}
```

InformationHandler branch 1's command

```
import lime.ILimeAgent;
import lime.LimeAgentMgr;
import lime.LimeThread;
import java.util.Vector;
import java.util.Iterator;
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class ClientPeer1BIH11Branch1Cmd extends LimeThread implements ILimeAgent{

public InformationHandler instance;

public ClientPeer1BIH11Branch1Cmd(InformationHandler p){
instance = p ;
}

public void run(){
//a temp Pair variable is declared in order to use primitive's return composed type (having more than one
    value)
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in
    the xml
Pair tempPair = null;

//empty operation
}

public LimeAgentMgr getMgr() {
return getParent().getMgr();
}
}
```

B.2.3 ClientPeer2's generated code

Root FaultHandler

```
import java.util.Vector;
import java.util.Iterator;
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class ClientPeer2BRootFH extends FaultHandler{
```

```
public ClientPeer2BRootFH instance;

public ClientPeer2BRootFH(SMEPPCaller cal){
instance = this;

super.setCatches(new Vector());
super.getCatches().add(new Catch(true,null));
this.execute(cal);
if(this.getContainer() instanceof Peer){
this.execute(this);
}
}

public void execute(SMEPPCaller cal){
super.setCaller(cal);
}

public void execute(ThreadedCommand parent) {

//a temp Pair variable is declared in order to use primitive's return composed type (having more than one
    value)
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in
    the xml
Pair tempPair = null;

ClientPeer2BDataTable.myCredentials = new Credential(new SMEPPKey("SMEPPKEY","SMEPPPWD"),
    new SMEPPKey[]{new SMEPPKey("TempReaderGroup","MyGroupNamePassword")});
peer.Primitives.getPrimitives().newPeer(ClientPeer2BDataTable.myCredentials,instance);
int i = 0;
while(i==0){
ClientPeer2BDataTable.gidArray = peer.Primitives.getPrimitives().getGroups(new GroupDescription("
    TempReaderGroup"),instance);
i = ClientPeer2BDataTable.gidArray.length;
try {Thread.sleep(1000);} catch (InterruptedException e) {e.printStackTrace();}
}
peer.Primitives.getPrimitives().joinGroup(ClientPeer2BDataTable.gidArray[0],ClientPeer2BDataTable.
    myCredentials,instance);
tempPair = peer.Primitives.getPrimitives().publish(ClientPeer2BDataTable.gidArray[0],new Contract("./
    ClientService2.xml"),instance);
ClientPeer2BDataTable.gsid = (GroupServiceID) tempPair.getFirst();
ClientPeer2BDataTable.psid = (PeerServiceID) tempPair.getSecond();
ClientPeer2BDataTable.temp = peer.Primitives.getPrimitives().invoke(ClientPeer2BDataTable.psid,"monitor"
    ,new Input(ClientPeer2BDataTable.gidArray[0]),instance);
//check if a fault has been raised during the waiting period of this invoke and returns if one has been raised
if(instance.isStopped()) return;
}

synchronized public void forwardFault(Fault f) {
super.setStopped(true);
boolean catched = false;
Iterator it = super.getCatches().iterator();
if(!catched && it.hasNext()){
catched=true;
Catch c = (Catch) it.next();
if(c.isAll() || c.getFault().getName().equals(f.getName())){
System.out.println(f.getName()+" caught");
//catch main command
System.exit(1);

}
}
}
}
```

DataTable

```
import peer.*;
```

```
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class ClientPeer2BDataTable {

private static ClientPeer2BDataTable instance;
public static Credential myCredentials;
public static GroupID tempGroupId;
public static Input temp;
public static GroupID[] gidArray;
public static PeerServiceID psid;
public static GroupServiceID gsid;

private ClientPeer2BDataTable(){

}

public static ClientPeer2BDataTable getDataTable(){
if(instance == null){
instance = new ClientPeer2BDataTable();
}
return instance;
}

}
```

B.2.4 ClientService's generated code

Root FaultHandler

```
import java.util.Vector;
import java.util.Iterator;
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class ClientService2RootFH extends FaultHandler{

public ClientService2RootFH instance;

public ClientService2RootFH(SMEPPCaller cal){
instance = this;

super.setCatches(new Vector());
super.getCatches().add(new Catch(true,null));
this.execute(cal);
if(this.getContainer() instanceof Peer){
this.execute(this);
}
}

public void execute(SMEPPCaller cal){
super.setCaller(cal);
}

public void execute(ThreadedCommand parent) {

//a temp Pair variable is declared in order to use primitive's return composed type (having more than one
    value)
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in
    the xml
Pair tempPair = null;
```

```
tempPair = peer.Primitives.getPrimitives().receiveMessage("monitor",instance);
//check if a fault has been raised during the waiting period of this receiveMessage and returns if one has been
    raised
if(instance.isStopped() || tempPair == null) return;
ClientService2DataTable.temp = (Input) tempPair.getSecond();
ClientService2DataTable.callerId = (CallerID) tempPair.getFirst();
ClientService2DataTable.gid = peer.Primitives.getPrimitives().getPublishingGroup(instance);
peer.Primitives.getPrimitives().subscribe("temp10s",ClientService2DataTable.gid,instance);
ClientService2IH11 ClientService2ih11 = new ClientService2IH11(instance);
peer.Primitives.getPrimitives().reply(ClientService2DataTable.callerId,"monitor",new Input(new String("
    anyOutputPossible")),instance);
}

synchronized public void forwardFault(Fault f) {
super.setStopped(true);
boolean catched = false;
Iterator it = super.getCatches().iterator();
if(!catched && it.hasNext()){
catched=true;
Catch c = (Catch) it.next();
if(c.isAll() || c.getFault().getName().equals(f.getName())){
System.out.println(f.getName()+" caught");
//catch main command
System.exit(1);

}
}
}
}
```

DataTable

```
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class ClientService2DataTable {

private static ClientService2DataTable instance;
public static CallerID callerId;
public static Input temp;
public static CallerID caller_temp;
public static Input temp10s;
public static GroupID gid;

private ClientService2DataTable(){

}

public static ClientService2DataTable getDataTable(){
if(instance == null){
instance = new ClientService2DataTable();
}
return instance;
}

}
```

InformationHandler's root

```
import java.util.Vector;
import java.util.Iterator;
```

```
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class ClientService2IH11 extends InformationHandler{

12 public ClientService2IH11 instance;

public ClientService2IH11(ThreadedCommand parent){
super.setUpperCmd(parent);
instance = this;
this.execute(super.getUpperCmd());
}

public void execute(ThreadedCommand parent) {

22 //a temp Pair variable is declared in order to use primitive's return composed type (having more than one
      value)
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in
      the xml
Pair tempPair = null;

ClientService2IH11Branch1 ClientService2ih11b1 = new ClientService2IH11Branch1(instance);
ClientService2ih11b1.start();
try{Thread.sleep(parser.SMoLParser.translateXPathDuration("PT30S"));}
catch(InterruptedException e){e.printStackTrace();}
peer.Primitives.getPrimitives().unsubscribe("temp10s",ClientService2DataTable.gid,instance);

32 super.setFinished();
}
}
```

InformationHandler branch 1

```
import lime.ILimeAgent;
import lime.LimeAgentMgr;
import lime.LimeThread;
import java.util.Vector;
import java.util.Iterator;
 6 import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class ClientService2IH11Branch1 extends LimeThread implements ILimeAgent {

public InformationHandler instance;
16
public ClientService2IH11Branch1(InformationHandler p){
instance = p;
}

public void run(){
//a temp Pair variable is declared in order to use primitive's return composed type (having more than one
      value)
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in
      the xml
Pair tempPair = null;

26 while(!instance.isFinished() || !instance.isStopped()){
//translation of the receiveEvent
tempPair = peer.Primitives.getPrimitives().receiveEvent(ClientService2DataTable.gid,"temp10s",instance);
```

```
//check if a fault has been raised during the waiting period of this receiveEvent and returns if one has been raised
if(instance.isStopped() || tempPair == null) return;
ClientService2DataTable.temp10s = (Input) tempPair.getSecond();
ClientService2DataTable.caller_temp = (CallerID) tempPair.getFirst();

//check if the informationHandler main command isn't terminated or if a fault was raised
if(instance.isFinished() || instance.isStopped()) return;

//starts the thread which executes the receiveEvent's command
ClientService2IH11Branch1Cmd ClientService2ih11b1cmd = new ClientService2IH11Branch1Cmd(instance);
ClientService2ih11b1cmd.start();
}
}

public LimeAgentMgr getMgr() {
return getParent().getMgr();
}
}
```

InformationHandler branch 1's command

```
import lime.ILimeAgent;
import lime.LimeAgentMgr;
import lime.LimeThread;
import java.util.Vector;
import java.util.Iterator;
import peer.*;
import peer.managers.services.*;
import sMoLTranslated.*;
import utils.*;
import utils.exceptions.*;
import utils.ids.*;

public class ClientService2IH11Branch1Cmd extends LimeThread implements ILimeAgent{

public InformationHandler instance;

public ClientService2IH11Branch1Cmd(InformationHandler p){
instance = p ;
}

public void run(){
//a temp Pair variable is declared in order to use primitive's return composed type (having more than one value)
//it is a temporary variable which purpose is to save the primitive's return value into the output specified in the xml
Pair tempPair = null;

//empty operation
}

public LimeAgentMgr getMgr() {
return getParent().getMgr();
}
}
```

B.3 Screen captures

B.3.1 TempReaderPeer

B.3.2 ClientPeer1

B.3.3 ClientPeer2

Appendix C

CoMA paper

Part of Chapter 4 has been published as a paper "*Secure P2P programming on top of tuple spaces*" at the *Workshop on Coordination Models and Applications: Knowledge in Pervasive Environments* (CoMA) held at the 17^{th} *IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises* (June 2008). We presented this paper in Rome on June 2008, the 24^{th}. The paper is given in the following six pages.

Secure P2P programming on top of tuple spaces*

F. Benigni[1], A. Brogi[1], J.L. Buchholz[1,2], J.M. Jacquet[2], J. Lange[1,2], R. Popescu[1]
(1) Department of Computer Science, University of Pisa, Italy
(2) Department of Computer Science, University of Namur, Belgium

Abstract

A new programming model for secure (embedded) peer-to-peer systems has been recently proposed in the context of the European project SMEPP. In this paper we present the design and implementation of such a model on top of tuple spaces. More precisely, we show how the SMEPP service-oriented interaction primitives can be effectively implemented using SecureLime.
Keywords: Peer-to-peer systems, tuple spaces.

1 Introduction and motivations

The flexibility of the peer-to-peer model allows the design of scalable and robust applications in many situations where a client-server approach is not suited, especially in those situations provided by mobile ad hoc networks where devices dynamically join and leave networks whose topology is not known in advance. The drawback of such a flexibility is a harder management of device discovery and coordination. However, since many low-level issues occurring in developing peer-to-peer systems are recurrent, various middleware solutions (e.g., JXTA [9]) have been deployed in order to ease the development of peer to peer applications by abstracting from those issues. One such middleware, especially targeted at enabling secure peer to peer communication between embedded systems, is currently under development in the SMEPP (Secure Middleware for Embedded Peer-To-Peer Systems) European project [3].

The SMEPP middleware is based on a service-oriented model allowing a dynamic integration of functionalities as devices get connected to the network. It is focused on security, so that SMEPP services will be easily provided and used with security guarantees that would be hard and inconvenient to achieve at application level.

To experiment the effectiveness of the interaction model designed for the SMEPP middleware, we have developed a prototype implementation of its functionalities. In this paper, we describe the design and implementation of such a prototype which has been built on top of tuple spaces. The idea of experimenting whether tuple spaces can provide a suitable basis to implement the SMEPP middleware has two main motivations. On the one hand, the abstract coordination model featured by tuple spaces has proven to notably ease the specification of complex distributed heterogeneous systems. On the other hand, the generative communication featured by tuple spaces can be seen as an enhancement of the basic coordination mechanism offered by standard data-centric storage techniques (e.g., distributed hash tables), which are indeed one of the key techniques employed in peer-to-peer systems

The SMEPP features introduce new coordination challenges since we have to cope with service and group availability, random peer disconnection, etc. without a central entity. As we will show in the following, the result of our work shows how a coordination language such as SecureLime [7] provides several mechanisms which make the implementation of a realistic service model much easier.

Roughly speaking, the two main issues to be faced in order to implement the SMEPP model on top of tuple spaces are: (1) how to express SMEPP service-oriented aspects (e.g., groups, services, communication a.s.o.), and (2) how to suitably implement security aspects. The tuple space based language we chose to implement the system is SecureLime [7], an extension of the Lime coordination language [11], that adds security properties to tuples and (federated) tuple spaces. As we will discuss in Section 3, we exploited Lime features (such as generative communication and federated tuple spaces) to implement service-oriented aspects, and SecureLime specific features to implement security aspects.

In the following we present the key concepts of the SMEPP model and an example of its use (Section 2), the design and implementation of the tuple space based prototype (Section 3), and some concluding remarks (Section 4).

*Work partly supported by the SMEPP project EU-FP6-IST 0333563.

2 The SMEPP model for secure P2P programming

We first introduce the key SMEPP concepts and the abstract SMEPP primitives followed by an example.

```
// Peer Management
pId newPeer(creds)
pId getPeerId(id?)
pId[] getPeers(gId)
// Group Management
gId createGroup(grDescr) gId[] getGroups(grDescr?)
grDescr getGroupDescription(gId)
void joinGroup(gId, creds)
void leaveGroup(gId)
// Service Management
<gSId, pSId> publish(gId, contract)
void unpublish(pSId)
<gId, gSId, pSId>[]
–getServices(gId?, pId?, sContract?, maxRes?, creds)
sContract getServiceContract(id)
sessId startSession(sId)
// Message Handling
out? invoke(eId, opName, in?)
<cId, in?> receiveMessage(gId?, opName)
void reply(cId, opName, out?, fName?)
// Event Handling
void subscribe(evName?, gId?)
void unsubscribe(evName?, gId?)
void event(gId?, evName, in?)
<cId, in?> receiveEvent(gId?, evName)
```

Figure 1. SMEPP Primitives.

Key SMEPP Concepts and Primitives. The analysis of current state-of-the-art models in P2P systems (see [1, 2, 4, 8, 9, 10]) reveals the fact that existing frameworks for the development of P2P applications generally (i) do not provide a simple, high-level service (interaction) model that presents a suitable level of abstraction to ease the development of P2P applications, or (ii) do not model key concepts such as group-wise security, services offered both by peers and groups, message and event-based communication.

The SMEPP service-oriented model aims to tackle such limitations. It features a set of abstract primitives (see Figure 1), which can be used to develop P2P application specifications in a simple, high-level manner. We aim at deploying such primitives as different (language dependent) APIs, which will allow the deployment of SMEPP specifications as real (executable) applications. The key SMEPP concepts are:

Peers. Roughly, peers are service containers. A peer executes a peer program built using the SMEPP primitives, and it may create or join groups, and offer or invoke services, as well as raise and receive events inside groups.

Groups. Groups are logical associations of peers, and they provide a secure communication environment, and a scope for published services. The SMEPP model offers security-aware primitives for group creation and joining. Furthermore, all communications among peers and services (see below) take place inside groups.

Services. Services have contracts and implementations. On the one hand, a contract provides descriptive information on a service (e.g., what the service does). On the other hand, the implementation is the executable service (e.g., a Java service). Peers publish services in groups. Furthermore, service clients (viz., peers or other services) join groups and either directly invoke a particular service provider, or blindly invoke a group service. Furthermore, services could invoke other services or peers and raise or receive events.

Communication Abstractions. Peers and services communicate by exchanging (data or fault) *messages*, or *events*. Messages are used as input and output (possibly empty) for services operations. On the one hand peers and services raise events, on the other hand other peers and services can subscribe to events of their interest and wait to receive them.

For further details on the SMEPP model please, see [3].

Temperature Monitoring Example. We present here a simple example of using the primitives to model the behaviour of both peers and services. Roughly, the example aims to describe the message-based interaction between peers and services, and it shows (i) how to create peers and groups, publish services, join groups, and (ii) how to directly invoke peer services, and how invocations of request-response operations behave.

Suppose that a `TempReaderPeer` peer creates a `TempReaderGroup` group in which it publishes a `TempReaderService` service. `TempReaderService` defines a `getTemp` request-response operation without input parameters, which (measures and) returns the ambient temperature. Another peer, `InvokerPeer`, joins the `TempReaderGroup` group, and then invokes the `getTemp` operation of `TempReaderService`. Using the SMEPP primitives one could implement the above scenario as follows.

TempReaderPeer. The top of Figure 2 presents the behaviour of `TempReaderPeer` using a pseudocode-like notation. We use the `opaque` keyword to hide the value assigned to a variable. `TempReaderPeer` first registers itself as a new SMEPP peer, and then it creates the `TempReaderGroup` group. Following, it publishes `TempReaderService` in `TempReaderGroup`, and then it continues processing (e.g., it could loop forever). Note that the termination of the peer's code implies the termination of `TempReaderService`, which is then unpublished by the middleware from `TempReaderGroup`.

TempReaderService. The middle of Figure 2 presents the behaviour of a state-less `TempReaderService`. `TempReaderService` first waits to receive an invocation of the `getTemp` operation. Then, it measures the temperature, and afterwards it replies to the invoker of `getTemp`. The execution of `TempReaderService` terminates after the reply.

InvokerPeer. The bottom of Figure 2 presents the behaviour of `InvokerPeer`. `InvokerPeer` first regis-

```
// TempReaderPeer
myCredentials = opaque;
newPeer(myCredentials);
mySecurityInfo = opaque;
myGroupDescription = <"TempReaderGroup",mySecurityInfo>;
tempGroupId = createGroup(myGroupDescription);
tempReaderServiceContract = opaque;
publish(tempGroupId, tempReaderServiceContract); ...
//TempReaderService
<callerId> = receiveMessage("getTemp");
temp = opaque;
reply(callerId, "getTemp", temp);
//InvokerPeer
myCredentials = opaque;
newPeer(myCredentials);
desiredGroupDescription = <"TempReaderGroup">;
gid[] = getGroups(desiredGroupDescription);
tempReaderServiceContract = opaque;
<groupId,tempReaderServiceGroupServiceId,
tempReaderServicePeerServiceId>[] = getServices(gid[0], tempReader-
ServiceContract, myCredentials);
joinGroup(groupId[0], myCredentials);
temp = invoke(tempReaderServicePeerServiceId[0], "getTemp");
```

Figure 2. `TempReaderPeer`.

ters itself as a new SMEPP peer, and then it discovers the group identifier of `TempReaderGroup` (viz., gid[0]), as well as the peer service identifier of `TempReader-Service`, which we assume it has been previously published by `TempReaderPeer` into `TempReaderGroup` (viz., tempServicePeerServiceId[0][1]). We assume for simplicity that `InvokerPeer` (partially) knows the contract of `TempReaderService`. Next, `InvokerPeer` joins `TempReaderGroup`, and then it invokes the `getTemp` operation of `TempReaderService`. This invocation blocks waiting for the result of `getTemp` because `getTemp` is a request-response operation. `InvokerPeer` terminates after the invocation returns.

3 Implementing P2P systems with SecureLime

As discussed in Section 2, the SMEPP models relies on messages and events. Linda-like models, based on shared data, can easily incorporate these features. This is particularly the case for the SecureLime model we have chosen. To sustain this fact, Subsection 3.1 and Subsection 3.2 first state the requirements for the implementation and explain why SecureLime is an appropriate target language. Subsection 3.3 then provides an overview of our implementation.

3.1 Requirements

Following Section 2, the main requirements on the target language are as follows. **[R1: Peer-to-peer orientation]** As the SMEPP project is by essence peer-to-peer and embedded devices oriented, we had to choose a language amenable to distributed implementations capable of managing transient connection of peers. **[R2: Available implementation]** The purpose of our work was to implement the SMEPP service model by using a Linda-like language, not to make a new implementation of an existing language. Having an executable language gives us the opportunity to prove the service model is usable in a real environment. **[R3: Security]** SMEPP defines a configurable model of security which, by default, uses symmetric keys to manage access rights: one preshared symmetric key to access a SMEPP application (i.e. to become a SMEPP peer) and one preshared symmetric key for each group. In addition to access control, SMEPP defines group and service visibility restrictions. A peer can only discover groups and services of which it has the corresponding key. The use of preshared symmetric keys prevents the implementation to be able to expel maliciously behaving peers (in other words, authorised peers are assumed to be well behaved during its whole life). **[R4: Java-integrability]** Since the reference implementation of the SMEPP project will be Java-based, we decided to develop our implementation using Java, in order to have useful feedback from our proof-of-concept. So we needed a middleware providing Java integrability.

3.2 Choice of a Linda-based language, SecureLime

Our first concern was to choose a convenient language on top of which to develop an implementation for the SMEPP service model. Our approach was to firstly identify the fundamental requirements, then we explored various well-known Linda-like languages to make our final choice. Our analysis lead us to choose SecureLime[7].

Lime (Linda In a Mobile Environment [11]) is a model and a middleware extending Linda with transiently shared tuple spaces. For instance, let peer A and peer B have both two shared tuple spaces, A owns X and Y, while B owns Y and Z. As soon as A and B get connected, the two instances of Y are merged into a *federated* one because they have the same name. Mainly the goal of Lime is to provide an application framework to coordinate mobile agents, which is of great interest in the SMEPP context. SecureLime[7] extends Lime by adding security properties to it. It adds access control on both tuple space and tuple level.

[R1] SecureLime perfectly meets the first requirement [R1] since it provides a completely decentralised architecture where every agent is only responsible for its shared data. This means that when disconnecting, agent's data gets unavailable to others, but the rest of the system stays unchanged. It fits really well with the offering and discovery of services in SMEPP. For instance, if a service offered by a peer is represented by a tuple in its Interface Tuple Space[2],

[1] TempServicePeerServiceId[0] denotes the value of tempServicePeer-ServiceId in <groupId, tempServiceGroupServiceId, tempServicePeer-ServiceId, serviceContract>[0].

[2] An ITS is the agent's local part of the federated tuple space.

in case of disconnection this tuple will become unavailable for others, so no peer can discover this service anymore. This feature also simplifies the group management in SMEPP as we will explain in the following subsection.

[R2] SecureLime provides an open-source implementation, complete and well-documented. This makes it a good candidate for our purposes since our aim is to avoid reinventing the wheel and implement a secure tuple space model from scratch. Moreover, one of the interesting features provided by the implementation is a fake GPS allowing to simulate physical disconnections of peers. As easily noted by the reader, these features meet requirement R2.

[R3] SecureLime offers two levels of access right control, one restricting access to tuple spaces and the other restricting access to tuples. The federation mechanism of Lime is modified in such a way that two tuple spaces are merged only if they were created using the same name *and* the same password. At the tuple level, SecureLime provides two special tuple fields, P_r and P_w: one enabling a peer to read the tuple only if it knows the P_r field value and the other one enabling a peer to remove the tuple only if it knows the P_w field value. The former turned out to be useful for managing access to the SMEPP world and groups. The latter is used to manage visibility of groups and services. With regards to the SMEPP security requirement [R3], SecureLime is fitting really well because it offers password based encryption which can be mapped directly to the SMEPP symmetric keys. This kind of security made the authentication of peers into the SMEPP application really easy, as we will explain in details later.

[R4] Furthermore, the implementation is entirely Java-based, which meets the last requirement [R4].

Some other Linda-like models and middlewares have been considered as potential candidates. Some of them were not implemented (e.g, SecSpaces), and so, even if interesting got automatically discarded. Other implemented models were discarded because they failed to meet some requirements (e.g., both JavaSpaces and TSpace rely on a centralised architecture, provide limited security and are not open source).

3.3 Overview of the implementation

We will describe here the key concepts of our implementation. This will be done in three steps. The first one shows how the SMEPP basic concepts are modelled using Lime ones. This leads to the second step, which enriches this model with security by using SecureLime security primitives. Finally, we will present two typical use cases of the primitives with a scenario based on the example in Section 2.

High-level design. An important part of SMEPP is the discovery of services and groups. In our implementation, this is done by using two tuple spaces, playing the role of service (*SD*) and group (*GD*) directories. The tuples contained in these two tuple spaces constitute respectively a list of service descriptions and a list of group descriptions. So, to search for a group or a service we use a *read* operation on the right tuple space.

A peer is mapped to a Lime Agent[3] which has two default tuple spaces[4] : the *SD* and the *GD* tuple spaces described above. When a peer wants to create or join a group *G*, it has to: (i) create a group tuple space *G*, (ii) put a tuple describing *G* in *GD*, and (iii) put a tuple in *G* to update the group membership list. A group is a set of peers which have executed these three actions. The members of a same group *G* use the group tuple space to communicate, as it will be illustrated in the scenarios below. A service can be discovered through its description-tuple[5] (providing its contract and IDs). As for the service implementation, a service is simply a Java-based process using our API. Basically, an event is modelled by the release of a tuple in a group tuple space *G*. In order to receive an event (`receiveEvent(.)` primitive), a peer creates a new Lime reaction [11] waiting for a tuple corresponding to the right event-tuple.

Security design. The security aspects of our implementation have been addressed by using the SecureLime's extensions of Lime. According to the SMEPP guidelines: (i) every peer has an AppKey password granting access to a SMEPP application, (ii) every peer has a set of passwords (GKeys) granting access to groups.

In order to prevent illegal peers to get access to the SMEPP application, the directory tuple spaces (*SD* and *GD*) are protected using SecureLime secured tuple spaces with AppKey as password. This way, every data passing through these tuple spaces is encrypted with AppKey.

Note that if the peer does not provide the right password at peer creation (`newPeer(.)` primitive), it will not get a fault message. Actually, it will create isolated directories (since the secured tuple spaces do not merge if they do not have the same password, thanks to the SecureLime federation mechanism). Furthermore, if an illegal peer creates a group, legal peers will not be able to see it since they will not share the same directories.

To ensure that every peer sees only groups and services matching its credentials, we had to prevent the tuples inside the directories from being visible to everyone. SecureLime made this task pretty easy. It suffices to use the password of the group as read-password, so a peer is only able to see a

[3] A Lime Agent is a Java-based process using the Lime API [11].
[4] Created by the execution of `newPeer(.)`.
[5] Which is placed after a `publish(.)` call in *SD*.

group or service description if it has the password matching the group or service visibility.

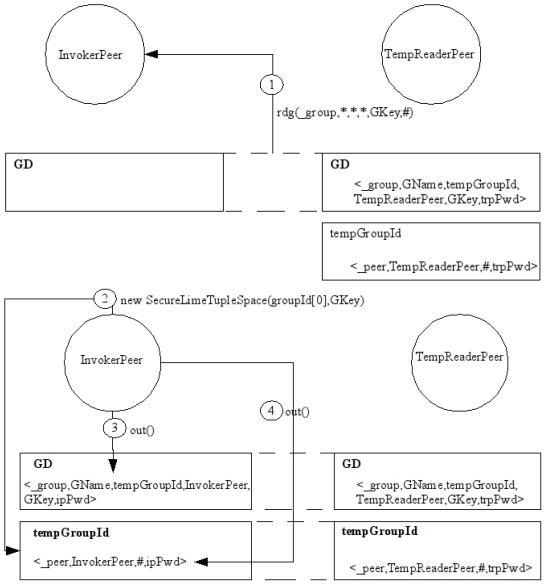
To restrict the access to a group, we protect the secured tuple space representing the group with the GKey corresponding to it. So when a peer tries to join a group, if it provides the right password, the newly created tuple space will be merged with the federated one. Otherwise, it will get an empty tuple space, as in the case of a `newPeer(.)` call with an incorrect password.

3.3.1 Scenarios.

The following scenarios illustrate how to map parts of the example found in Section 2 to the SecureLime API. Firstly, we will describe how the `getGroups(.)` and the `joinGroup(.)`, which are called by *InvokerPeer*, work. Then we will describe how the invocation of the *getTemp* operation is translated in our API.

The first scenario makes the assumptions that the peers share the AppKey and the password to get access to *tempGroupId*, *GKey* in the following. In Figure 3, only one group (*tempGroupId*) has been created (by *TempReaderPeer*), so a tuple referencing it exists in *GD*. This tuple contains the name of the group, its ID and the reference owner. It is read-protected by *GKey* and remove-protected by *trpPwd*[6]. There is also a membership-tuple for *TempReaderPeer* in *tempGroupId*. This one contains the peer's ID[7].

The second scenario illustrates the invocation of the *TempReaderService* service. Figure 4 shows that both peers are members of the *tempGroupId* group since they have the *tempGroupId* tuple space, the reference-tuple and the membership-tuple. *TempReaderPeer* has published the *TempReaderService* service, so there is a tuple in *SD* containing the service contract, the peer service ID (*TRS-PSID*), the group service ID (*TRS-GSID*), the group in which it has been published and the provider. It is read-protected by *GKey* and remove-protected by *trpPwd*, a password generated by *TempReaderPeer*.



1. *InvokerPeer* invokes `getGroups(desiredGroupDescription, myCredentials)` to discover which groups it can join. To do this, it performs a read operation on the federated *GD* tuple space using *GKey* as read-password. *InvokerPeer* gets the only tuple containing the group name and its ID which are the needed information to join the group later.
2. *InvokerPeer* invokes `joinGroup(groupId[0],myCredentials)` which creates the new SecureLimeTupleSpace object, representing the *tempGroupId* group tuple space. The peer has to use the group ID (*groupId[0]*) and the password corresponding to it (*GKey*). Since the two *tempGroupId* tuple spaces have the same name and password, they automatically merge to create a federated tuple space.
3. `joinGroup(.)` continues by putting a reference-tuple describing the group in the *GD* tuple space. This step ensures that the group will be kept alive as long as the group contains at least one member. This tuple differs from *TempReaderPeer*'s one by the remove-password field (*ipPwd*) and the field expressing the tuple owner.
4. The last step of `joinGroup(.)` consists of putting the peer "membership-tuple" into the *tempGroupId* tuple space.

Figure 3. Group management.

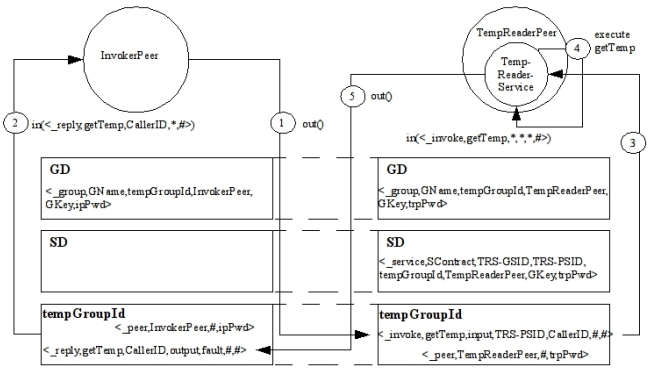
4 Concluding Remarks

In this paper we described a running prototype implementation of the SMEPP primitives using SecureLime [7]. We consider the experiment was successful as the implementation complies with the SMEPP objectives and requirements [12]. Furthermore, this work showed how a real world service model can be implemented using a tuple space coordination language. We argue the paper features the following main contributions. Firstly, we provide an extension of SecureLime with the SMEPP programming model that offers a simple high-level API supporting the definition of: peer and service code, peer groups, group-wise security, synchronous and asynchronous message patterns (one-to-one, direct or blind operation invocations), event-based communication (one-to-many). Then, we have created a proof-of-concept prototype implementation of the SMEPP model that can be used to test/simulate interactions of peers and services. This implementation differs from the ones presented in [6] and [5], mainly because our service model features many more coordination concepts. In [6] and [5] only client, server and service are offered, while in the SMEPP service model groups, events and sessions are provided. However, service discovery is handled in a similar way, using a tuple space as repository.

We selected SecureLime for implementation based on the assessment of several tuple space-based coordination models with respect to SMEPP key requirements. SecureLime fulfills such requirements, and it also provides features such as federated (password-protected) tuple spaces, or read/remove tuple passwords. These features allowed us

[6]The last two fields of every tuple are respectively the read and remove-password. Here *trpPwd* (which has been securely generated by the peer) is used to ensure that only the tuple's owner can remove it.

[7]The symbol # means that no password is used.



1. *InvokerPeer* calls `invoke(tempReaderServicePeerServiceId[0],''getTemp'')` which firstly puts an "invocation-tuple" into *TempReaderPeer*'s local *tempGroupId* tuple space. This tuple contains the operation name (`getTemp`) and its parameters (empty here), the service ID (here the peer service ID contained in `tempReaderServicePeerServiceId[0]`) and the ID of the caller.
2. Since `invoke(.)`'s execution must be blocked until the provider has done a `receiveMessage(.)`, *InvokerPeer* will perform a (blocking) *in* operation, waiting for the "reply-tuple" related to the *getTemp* operation.
3. When `receiveMessage(groupId,"getTemp")` is called by *TempReaderService*, it retrieves an "invocation-tuple" from the local tuple space of its container (*TempReaderPeer*) by doing a *in* operation on it. The template of this operation contains only the operation name (*getTemp*).
4. Here the service actually executes the operation.
5. *TempReaderService* calls `reply(.)` which puts a "reply-tuple" into the local *tempGroupId* tuple space of *InvokerPeer* (*getTemp* caller). This tuple contains the operation name, the caller ID, the operation result and a possible fault. This last action will unblock the invoker's execution.

Figure 4. Service invocation.

to successfully model all SMEPP key concepts. However, a limitation of SecureLime is that it does not provide a way to change "on-the-fly" the tuple space passwords. Consequently, the developer has to deal with this issue at the application level.

We have also defined a SMEPP specification language (SMoL [3]), which allow one to orchestrate SMEPP primitives into complex behaviour (e.g., using sequential, parallel, choice, or event and fault handler operators). SMoL is meant to assist the SMEPP developer into (semi-automatically) generating peer or service code. Furthermore, such a language enables the formal analysis of the behaviour of peers and services, and of their interactions [3]. We have implemented a SMoL2Java translator (which generates Java code from a SMoL specification) and we have integrated it with our implementation based on SecureLime. The resulting prototype produces executable Java code that runs on top of SecureLime starting from a SMoL description of the behaviour of a peer or service[8]. Unfortunately, space limitations do not allow us to describe the details of the translation here.

Beyond the above mentioned limitations due to the restrictions of the current SecureLime release, our prototype only features a basic mechanism for the discovery of service contracts based on the syntactic matching of tuples.

Our planned next step is to thoroughly experiment the prototype in order to engineer the implementation. We also intend to overcome the present limitations, and to experiment the implementation of an enhanced security mechanism based on session keys.

[8]The proof-of-concept prototype can be downloaded from http://www.smepp.org/Downloads.aspx. Note however that the usage of the prototype requires an understanding of SMoL and of the SMoL2Java translator.

References

[1] S. Alda and A. Cremers. Towards composition management for component-based peer-to-peer architectures. *ENTCS*, 114:47–64, 2005.

[2] M. Bisignano, G. Modica, and O. Tomarchio. Jmobipeer: A middleware for mobile peer-to-peer computing in manets. In *ICDCS'05*, pages 785–791, 2005.

[3] A. Brogi, R. Popescu, F. Gutierrez, P. Lopez, and E. Pimentel. A service-oriented model for embedded peer-to-peer systems. In *FOCLASA'07, ENTCS*, 2007.

[4] G. Gehlen and L. Pham. Mobile web services for peer-to-peer applications. In *CCNC'05*, pages 427–433, 2005.

[5] R. Handorean and G.-C. Roman. Service provision in ad hoc networks. In *Coordination Models and Languages*, LNCS, pages 207–219, 2002.

[6] R. Handorean and G.-C. Roman. Secure service provision in ad hoc networks. In *ICSOC'03*, volume 2910 of *LNCS*, pages 367–383, 2003.

[7] R. Handorean and G.-C. Roman. Secure sharing of tuple spaces in ad hoc settings. *ENTCS*, 85(3), 2003.

[8] R. Handorean, R. Sen, G. Hackmann, and G.-C. Roman. Supporting predictable service provision in manets via context aware session management. *JWSR*, (3):1–26, 2006.

[9] JXTA. `http://www.jxta.org/`.

[10] R. Lucchi and G. Zavattaro. Wssecspaces: a secure data-driven coordination service for web services applications. In *SAC'04*, pages 487–491. ACM, 2004.

[11] A. L. Murphy, G. P. Picco, and G.-C. Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM TOSEM*, 15(3):279–328, 2006.

[12] SMEPP Coalition. D1.1: State of the art and generic middleware requirements. `www.smepp.org`.

[13] SMEPP Coalition. D1.2: Security requirements of EP2P applications. `www.smepp.org`.